An Iterated Local Search Algorithm for the Time-Dependent Vehicle Routing Problem with Time Windows*

Hideki Hashimoto†, Mutsunori Yagiura‡, Toshihide Ibaraki§

June 30, 2007

Abstract

We generalize the standard vehicle routing problem with time windows by allowing both traveling times and traveling costs to be time-dependent functions. In our algorithm, we use a local search to determine routes of the vehicles. When we evaluate a neighborhood solution, we must compute an optimal time schedule of each route. We show that this subproblem can be efficiently solved by dynamic programming, which is incorporated in the local search algorithm. The neighborhood of our local search consists of slight modifications of the standard neighborhoods called 2-opt*, cross exchange and Or-opt. We propose an algorithm that evaluates solutions in these neighborhoods more efficiently than computing the dynamic programming from scratch by utilizing the information from the past dynamic programming recursion used to evaluate the current solution. We further propose a filtering method that restricts the search space in the neighborhoods to avoid many solutions having no prospect of improvement. We then develop an iterated local search algorithm that incorporates all the above ingredients. Finally we report computational results of our iterated local search algorithm compared against existing methods, and confirm the effectiveness of the restriction of the neighborhoods and the benefits of the proposed generalization.

1 Introduction

The *vehicle routing problem* (VRP) is the problem of minimizing the total travel distance of a number of vehicles, under various constraints, where every customer must be visited exactly once by a vehicle [1, 2, 3]. This is one of the representative combinatorial optimization problems and is known to be NP-hard. The traveling salesman problem (TSP) can be viewed as a special case of VRP in which the number of vehicles is one. TSP has been intensively studied for many decades [4] and VRP is under intensive study as well. Among variants of VRP, the VRP with capacity and time window constraints, called the *vehicle routing problem with time windows* (VRPTW), has been widely studied in the last decade [5, 6, 7, 8]. The capacity constraint signifies that the total load on a route cannot exceed the capacity of the assigned vehicle. The time window constraint signifies that each vehicle must start the service at each customer in the period specified by the customer. The VRPTW has a wide range of applications such as bank deliveries, postal deliveries, school bus routing and so on, and has been a subject of intensive

*Department of Applied Mathematics and Physics Technical Report 2007-13

†Department of Applied Mathematics and Physics, Graduate School of Informatics, Kyoto University (hasimoto@amp.i.kyoto-u.ac.jp)

‡Department of Computer Science and Mathematical Informatics, Graduate School of Information Science, Nagoya University

§Department of Informatics, School of Science and Technology, Kwansei Gakuin University

research focused mainly on heuristic and metaheuristic approaches. See extensive surveys by Bräysy and Gendreau [9, 10] for heuristic and metaheuristic approaches.

A constraint is called *hard* if it must be satisfied, while it is called *soft* if it can be violated. The violation of soft constraints is usually penalized and added to the objective function. The VRP with hard (resp., soft) time window constraints is abbreviated as VRPHTW (resp., VRPSTW). For VRPHTW, even just finding a feasible schedule with a given number of vehicles is known to be NP-complete in the strong sense, because it includes the (one-dimensional) bin packing problem as a special case [11]. Hence it may not be reasonable to restrict the search only within the feasible region of VRPHTW, especially when the constraints are tight. Moreover, in real-world situation, time window and capacity constraints can be often violated to some extent. Considering these, the two constraints (i.e., time window constraint and capacity constraint) are treated as soft in this paper. To evaluate constraint violation, we use cost functions, which can be non-convex and/or discontinuous as long as they are piecewise linear. This formulation is quite general; e.g., one or more time slots can be assigned to each customer.

In real situations, traveling times are often dependent on the departure times and they cannot be treated as constants in such cases (e.g., rush-hour traffic jam). For TSP, the generalization with time-dependent traveling times is called the *time dependent traveling salesman problem* (TDTSP) and is well-studied [12, 13, 14]. On the contrary, to the best of our knowledge, not much has been investigated on similar generalizations of VRPTW except for the work by Ichoua et al. [15]. In their formulation, each customer have only one time window. In this paper, we introduce traveling time and cost functions between each customer, whose values are dependent on the start time of traveling. These functions can be nonconvex and/or discontinuous as long as they are piecewise linear. Although we assume some property for each traveling time function, any functions satisfying the FIFO condition considered in [15] can still be represented, and the problem is fairly general. Our model generalizes that of Ichoua et al. in that it can allow more flexible time penalty function for each customer, and that of Ibaraki et al. [16] in that it can treat time-dependent traveling time and cost.

In our algorithm, we use local search to determine the routes of vehicles. When we evaluate a neighborhood solution, we need to solve the problem of determining the optimal start times on each route. In Ichoua et al., they solve this subproblem approximately (for their restricted formulation), but solve it exactly only for the best M approximate neighborhood solutions (M is a parameter). We show that this subproblem can be efficiently solved with dynamic programming. The time complexity of our dynamic programming algorithm is the same as that of Ibaraki et al. [16] in spite of its generality if each traveling time and cost are constants. This dynamic programming is incorporated in the local search algorithm. In our local search, we use the standard neighborhoods called 2-opt*, cross exchange and Or-opt with slight modifications. We can evaluate the solutions in these neighborhoods efficiently by utilizing the information from the past dynamic programming recursion. We further propose a filtering method to restrict the search in the neighborhoods to avoid many solutions having no prospect of improvement. For the 2-opt* neighborhood, even with this restriction, we will not miss a better solution in the neighborhood if there is any. We develop an iterated local search algorithm incorporating all the above ingredients. Finally we report computational results on benchmark instances, and confirm the effectiveness of the restriction of the neighborhood. We compare the performance of our iterated local search algorithm against existing methods, and discuss the benefits of the proposed generalization.

2 Problem

Here we formulate the time-dependent vehicle routing problem with time windows. Let $G = (V, E)$ be a complete directed graph with vertex set $V = \{0, 1, \ldots, n\}$ and edge set $E = \{(i, j) \mid i, j \in V, i \neq j\}$, and $M = \{1, 2, \ldots, m\}$ be a vehicle set. In this graph, vertex 0 is the depot and other vertices are customers. Each customer i, each vehicle k and each edge $(i, j) \in E$ are associated with:

i. a fixed quantity a_i (≥ 0) of goods to be delivered to i,

ii. a time window cost function $p_i(t)$ of the start time t of the service at i ($p_0(t)$ is the time window cost function of the arrival time t at the depot),

iii. a capacity u_k (≥ 0) of k,

iv. a traveling time function $\lambda_{ij}(t)$ and a traveling cost function $q_{ij}(t)$ from i to j when the start time is t.

We assume $a_0 = 0$ without loss of generality. Each time window cost function $p_i(t)$ is nonnegative, piecewise linear and lower semicontinuous (i.e., $p_i(t) \leq \lim_{\varepsilon \to 0} \min\{p_i(t+\varepsilon), p_i(t-\varepsilon)\}$ at every discontinuous point t). Note that $p_i(t)$ can be non-convex and discontinuous as long as it satisfies the above conditions. We also assume $p_i(t) = +\infty$ for $t < 0$ so that the start time t of the service is nonnegative. We assume that each traveling cost function $q_{ij}(t)$ satisfies the same conditions as $p_i(t)$ (i.e., nonnegative, piecewise linear, lower semicontinuous and $q_{ij}(t) = +\infty$ for $t < 0$). We assume that each traveling time function $\lambda_{ij}(t)$ is nonnegative, piecewise linear and lower semicontinuous. The number of linear pieces of these functions are assumed to be finite. These assumptions ensure the existence of an optimal solution. We further assume that $\lambda_{ij}(t)$ satisfies

$$\begin{aligned} & t + \lambda_{ij}(t) = t' + \lambda_{ij}(t') \\ & \Rightarrow t + \lambda_{ij}(t) = \alpha t + (1-\alpha)t' + \lambda_{ij}(\alpha t + (1-\alpha)t'), \quad 0 \leq \alpha \leq 1 \end{aligned} \tag{1}$$

unless otherwise stated (see an example in Figure 1). In Figure 1, $s = t + \lambda_{ij}(t)$ is the arriving time at j when a vehicle departs from i at t. It is known that the FIFO condition in [15] (i.e., $t \leq t' \Rightarrow t + \lambda_{ij}(t) \leq t' + \lambda_{ij}(t')$) implies condition (1). In our problem, the linear pieces of each piecewise linear function are given explicitly (i.e, the number of linear pieces is a part of the input size).

Let σ_k denote the route traveled by vehicle k, where $\sigma_k(h)$ denotes the hth customer in σ_k, and let

$$\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_m).$$

Note that each customer i is included in exactly one route σ_k, and is visited by vehicle k exactly once. We denote by n_k the number of customers in σ_k. For convenience, we define $\sigma_k(0) = 0$ and $\sigma_k(n_k + 1) = 0$ for all k (i.e., each vehicle $k \in M$ departs from the depot and comes back to the depot). Moreover, let s_i be the start time of service at customer i (by exactly one of the vehicles) and s^{a}_k be the arrival time of vehicle k at the depot, and let

$$\boldsymbol{s} = (s_1, s_2, \ldots, s_n, s^{\mathrm{a}}_1, s^{\mathrm{a}}_2, \ldots, s^{\mathrm{a}}_m).$$

We assume $s_0 = 0$ for convenience of explanation. Let t_i be the departure time of a vehicle from customer i and t^{l}_k be the departure time of vehicle k from the depot, and let

$$\boldsymbol{t} = (t_1, t_2, \ldots, t_n, t^{\mathrm{l}}_1, t^{\mathrm{l}}_2, \ldots, t^{\mathrm{l}}_m).$$

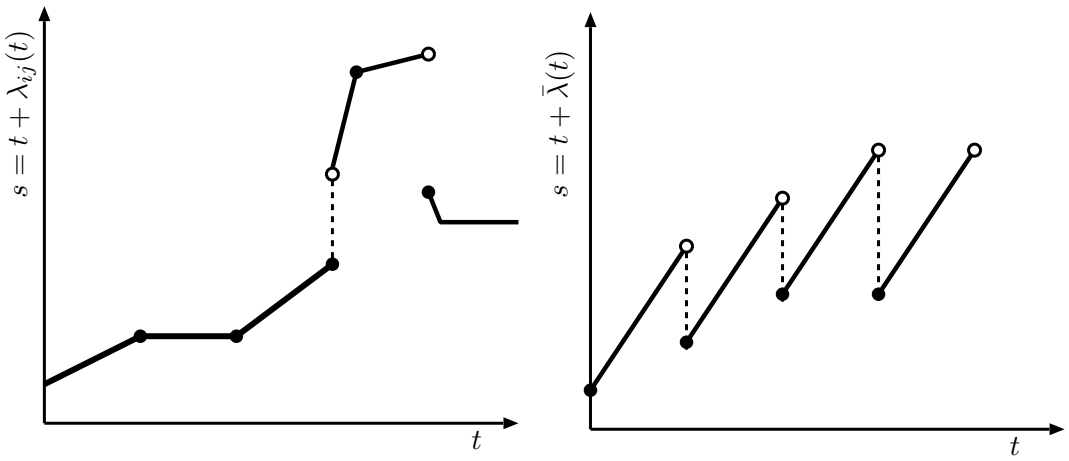


Figure 1: An example of λ_{ij} which satisfies condition (1), and a function $\bar{\lambda}$ which does not satisfy condition (1)

Note that each vehicle is allowed to wait at customers before starting services and before traveling.

Let us introduce 0-1 variables $y_{ik}(\boldsymbol{\sigma}) \in \{0,1\}$ for $i \in V \setminus \{0\}$ and $k \in M$ by

$$y_{ik}(\boldsymbol{\sigma}) = 1 \iff i = \sigma_k(h) \text{ holds for exactly one } h \in \{1, 2, \ldots, n_k\}.$$

That is, $y_{ik}(\boldsymbol{\sigma}) = 1$ if and only if vehicle k visits customer i. Then the total traveling cost q_{sum} traveled by all vehicles, the total time window cost p_{sum} for start times of services, and the total amount a_{sum} of capacity excess are expressed as follows:

$$\begin{aligned}
p_{\mathrm{sum}}(\boldsymbol{s}) &= \sum_{i \in V \setminus \{0\}} p_i(s_i) + \sum_{k \in M} p_0(s_k^{\mathrm{a}}), \\
q_{\mathrm{sum}}(\boldsymbol{\sigma}, \boldsymbol{t}) &= \sum_{k \in M} q_{0,\sigma_k(1)}(t_k^{\mathrm{l}}) + \sum_{k \in M} \sum_{h=1}^{n_k} q_{\sigma_k(h),\sigma_k(h+1)}(t_{\sigma_k(h)}), \\
a_{\mathrm{sum}}(\boldsymbol{\sigma}) &= \sum_{k \in M} \max \left\{ \sum_{i \in V} a_i y_{ik}(\boldsymbol{\sigma}) - u_k, 0 \right\}.
\end{aligned}$$

Then the problem we consider in this paper is formulated as follows:

$$\begin{aligned}
\text{minimize} \quad & cost(\boldsymbol{\sigma}, \boldsymbol{s}, \boldsymbol{t}) = p_{\mathrm{sum}}(\boldsymbol{s}) + q_{\mathrm{sum}}(\boldsymbol{\sigma}, \boldsymbol{t}) + a_{\mathrm{sum}}(\boldsymbol{\sigma}) && (2) \\
\text{subject to} \quad & \sum_{k \in M} y_{ik}(\boldsymbol{\sigma}) = 1, \qquad i \in V \setminus \{0\} && (3) \\
& s_i \le t_i, \qquad i \in V \setminus \{0\} && (4) \\
& t_k^{\mathrm{l}} + \lambda_{0,\sigma_k(1)}(t_k^{\mathrm{l}}) \le s_{\sigma_k(1)}, \qquad k \in M && (5) \\
& t_{\sigma_k(i)} + \lambda_{\sigma_k(i),\sigma_k(i+1)}(t_{\sigma_k(i)}) \le s_{\sigma_k(i+1)}, \\
& \qquad 1 \le i \le n_k - 1,\ k \in M && (6) \\
& t_{\sigma_k(n_k)} + \lambda_{\sigma_k(n_k),0}(t_{\sigma_k(n_k)}) \le s_k^{\mathrm{a}}, \qquad k \in M. && (7)
\end{aligned}$$

Constraint (3) means that every customer $i \in V \setminus \{0\}$ must be served exactly once by a vehicle. Constraint (4) requires that each vehicle must depart from customer i after the service and

constraints (5)–(7) require that each vehicle cannot serve a customer before arriving at the customer. The time window and capacity constraints are treated as soft, and their violation is evaluated as the costs $p_{\mathrm{sum}}(\boldsymbol{s})$ and $a_{\mathrm{sum}}(\boldsymbol{\sigma})$ in the objective function. Note that, for any solution with a finite cost, $t^{\mathrm{l}}_k \geq 0$ holds because of the assumptions, and hence $\boldsymbol{s}, \boldsymbol{t} \geq 0$ hold.

Although we assume that each service takes no time, we can treat the case with positive constant service times by defining each traveling time and cost functions as $\lambda_{ij}(t) = \tilde{b}_i + \tilde{\lambda}_{ij}(t+\tilde{b}_i)$ and $q_{ij}(t) = \tilde{q}_{ij}(t + \tilde{b}_i)$ if the given traveling time and cost functions between customer i and j are $\tilde{\lambda}_{ij}$ and $\tilde{q}_{ij}$, and the service time of customer i is $\tilde{b}_i$. In our formulation, a traveling cost function can be a constant function such as distance, which is a major objective function in traditional formulations, and hence our problem is a generalization of VRPSTW and the model of Ibaraki et al. [16].

This problem is separated into m scheduling problems of finding the optimal start times if vehicle routes $\boldsymbol{\sigma}$ are fixed. Hence our algorithm searches $\boldsymbol{\sigma}$ by local search and solve the corresponding m scheduling problems for each $\boldsymbol{\sigma}$ generated during the search. In Section 3, we discuss this scheduling problem. How to search σ_k will be discussed in Section 4.

3 Optimal start time problem

In this section, we consider the problem of determining the optimal start times for a given route σ_k so that the total cost is minimized. Since the route is given, the objective function we have to consider is the sum of the time window costs and traveling costs. We call this subproblem the *optimal start time problem* and abbreviate it as OSTP in this paper.

For convenience, throughout this section, we assume that vehicle k visits customers $1, 2, \ldots, n_k$ in this order. Let customer 0 represent the departure from the depot (i.e., $t_0 = t^{\mathrm{l}}_k$ and $q_{0,1}(t_0) = q_{0,1}(t^{\mathrm{l}}_k)$), and let customer $n_k + 1$ represent the arrival at the depot (i.e., $s_{n_k+1} = s^{\mathrm{a}}_k$ and $p_{n_k+1}(s_{n_k+1}) = p_0(s^{\mathrm{a}}_k)$).

Then, the OSTP is described as follows:

$$
\begin{array}{lll}
\text{minimize} & \displaystyle\sum_{h=1}^{n_k+1} p_h(s_h) + \sum_{h=0}^{n_k} q_{h,h+1}(t_h) & \\
\text{subject to} & s_h \leq t_h, & 1 \leq h \leq n_k \\
& t_h + \lambda_{h,h+1}(t_h) \leq s_{h+1}, & 0 \leq h \leq n_k.
\end{array}
$$

We can solve the OSTP by a dynamic programming algorithm in polynomial time as will be explained in Section 3.1.

3.1 Dynamic programming

We will show that the OSTP is solvable in polynomial time by using dynamic programming.

> Let $f_h(t)$ be the minimum sum of the time window costs for customers 0, 1, ..., h and the traveling costs between them under the condition that they are all served before time t.

We call $f_h(t)$ as a *forward minimum cost function*. Then it can be computed by the following recurrence formula of dynamic programming:

$$
f_0(t) \;=\; \begin{cases} +\infty, & t < 0 \\ 0, & t \geq 0 \end{cases}
$$

$$f_h(t) = \min_{s_h \le t} \left\{ p_h(s_h) + \min_{t_{h-1}:\ t_{h-1}+\lambda_{h-1,h}(t_{h-1}) \le s_h} \{ f_{h-1}(t_{h-1}) + q_{h-1,h}(t_{h-1}) \} \right\},$$
$$1 \le h \le n_k + 1, -\infty < t < +\infty. \quad (8)$$

The optimal cost of the OSTP for a route σ_k is given by $\min_t f_{n_k+1}(t)$.

3.2 Algorithm and time complexity

In this subsection, we consider the data structure and algorithm for computing forward minimum cost functions f_h in the recurrence formula (8). Since all functions of the input are piecewise linear, each f_h is also piecewise linear. We can therefore store all functions in linked lists; each cell stores the interval and the linear function of the corresponding piece, and the cells are linked according to the order of intervals. For example, Figure 2 shows a piecewise linear function g and the corresponding linked list.

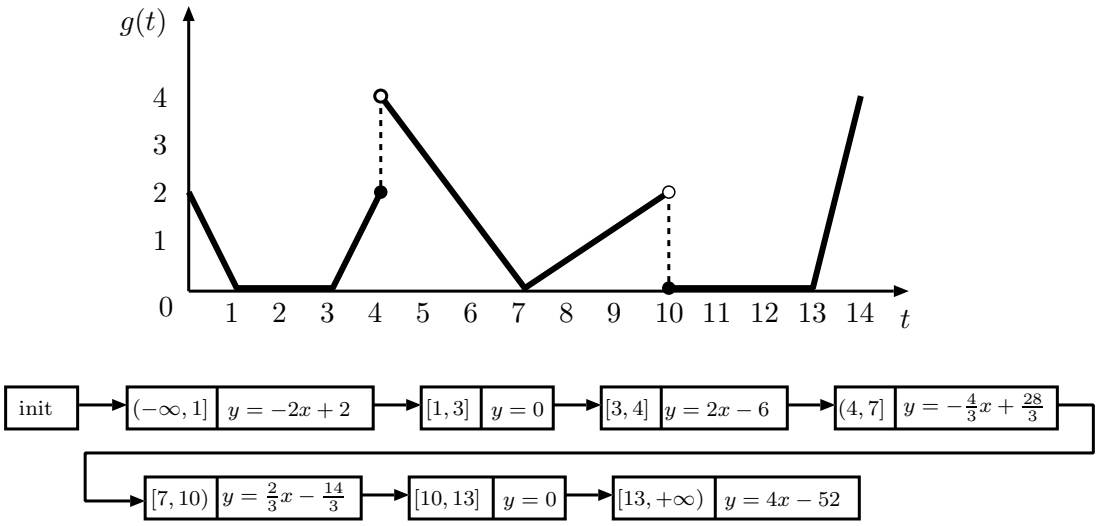


Figure 2: A function g and the linked list that represents g

Let $\delta(g)$ be the sum of the number of linear pieces and the number of discontinuous points of a piecewise linear function g (i.e., the number of pieces of the polygonal line of g). For example, the function g in Figure 2 has seven pieces and two discontinuous points, and hence $\delta(g) = 9$. Then it is straightforward to see that the summation $g + g'$ of two piecewise linear functions g and g' can be computed in $O(\delta(g) + \delta(g'))$ time and the resulting function satisfies $\delta(g + g') \le \delta(g) + \delta(g')$. It is also easy to see that function $\phi(t) = \min_{x \le t} g(x)$ can be computed in $O(\delta(g))$ time and the resulting function ϕ satisfies $\delta(\phi) \le \delta(g)$. When g is an increasing (resp., decreasing) piecewise linear function, i.e, $t < t' \Rightarrow g(t) < g(t')$ (resp., $t < t' \Rightarrow g(t) > g(t')$), we can compute the composite function $g' \circ g$ (i.e., $g' \circ g(t) = g'(g(t))$) for a piecewise linear function g' in $O(\delta(g) + \delta(g'))$ time since we can compute $g'(g(t))$ by increasing $g(t)$ gradually. In this case, the resulting function satisfies $\delta(g' \circ g) \le \delta(g) + \delta(g')$. We can also compute the inverse of g in $O(\delta(g))$ time, and the inverse g^{-1} satisfies $\delta(g^{-1}) = \delta(g)$.

We define

$$\gamma_h(s) = \begin{cases} \min\{ f_{h-1}(t) + q_{h-1,h}(t) \mid t + \lambda_{h-1,h}(t) = s \}, \\ \qquad\qquad \text{if } \{ t \mid t + \lambda_{h-1,h}(t) = s \} \neq \emptyset, \\ \infty \qquad\qquad \text{otherwise,} \end{cases}$$

and reformulate the above recurrence formula (8) as

$$f_h(t) \quad = \quad \min_{s_h \le t} \left\{ p_h(s_h) + \min_{s \le s_h} \gamma_h(s) \right\}. \tag{9}$$

To compute f_h by the recurrence formula (9), we must compute the function $\gamma_h(s)$ first. Let

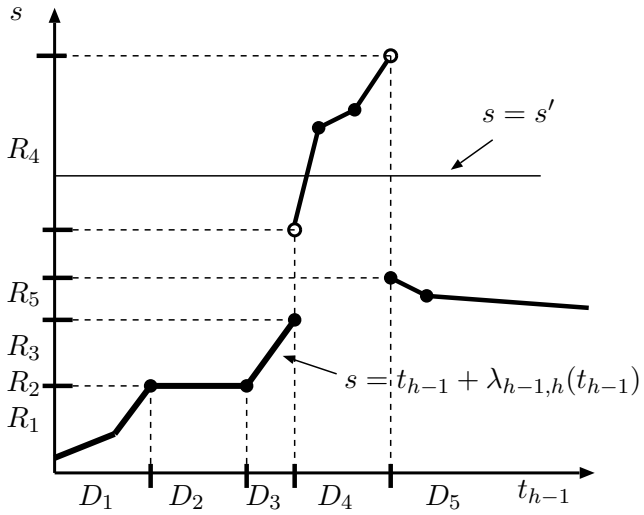


Figure 3: The relationship between the departure time t_{h-1} and the arriving time $s = t_{h-1} + \lambda_{h-1,h}(t_{h-1})$

us consider the plane whose horizontal axis corresponds to the start time t_{h-1} of traveling and the vertical axis corresponds to the arriving time $s = t_{h-1} + \lambda_{h-1,h}(t_{h-1})$. This is illustrated in Figure 3. Then $\gamma_h(s')$ for a fixed $s = s'$ is the minimum value of $f_{h-1}(t_{h-1}) + q_{h-1,h}(t_{h-1})$ among the points that satisfy

$$t_{h-1} + \lambda_{h-1,h}(t_{h-1}) = s', \tag{10}$$

if such a point exists. In order to compute $\gamma_h(s)$, we split the domain of t_{h-1} into increasing, constant and decreasing continuous parts and denote the closures of split intervals as $D_1, D_2, \ldots, D_L$ (see $D_1, D_2, \ldots, D_5$ in Figure 3). Then, for each $l = 1, 2, \ldots, L$, function $t + \lambda_{ij}(t)$ on domain D_l admits the inverse or is a constant function. Let R_l be the range of $t + \lambda_{ij}(t)$ on domain D_l (i.e., $R_l = \{t + \lambda_{ij}(t) \mid t \in D_l\}$). By condition (1) and the definition of D_l, $R_l \cap R_{l'}$ contains at most one point for any $l \neq l'$. Hence we can compute $\gamma_h(s)$ partially for each domain D_l except for $s \in \cup_{l \neq l'} R_l \cap R_{l'}$. Then $\gamma_h(s)$ is completed by merging them and taking the minimums for $s \in \cup_{l \neq l'} R_l \cap R_{l'}$. Note that if $s \notin \cup_{l=1}^{L} R_l$, we define $\gamma_h(s) = \infty$. We have to arrange all R_l's in the increasing order when we merge them, because the order of the appearance of R_l may be different from that of D_l. In Figure 3, the order of the appearance of R_l is $(R_1, R_2, R_3, R_5, R_4)$. However, the order of R_l's can be determined by λ_{ij} alone and need be computed only once before a search. This computation is negligible in comparison with the whole computation time. Hence we assume that the order of R_l's for each λ_{ij} is given as a part of input.

We can now compute $\gamma_h(s)$ by the following steps:

i. Compute $\gamma_h|_{R_l}$ for each domain $R_1, R_2, \ldots, R_L$ by increasing t_{h-1} gradually and computing the corresponding $\gamma_h(s)$.

ii. Merge $\gamma_h|_{R_l}$ for $l = 1, 2, \ldots, L$ and add linear pieces for the intervals with $s \notin \cup_{l=1}^{L} R_l$.

For computing each $\gamma_h|_{R_l}$ for $l = 1, 2, \ldots, L$, we need either to take the minimum (i.e., $\gamma_h(s) = \min\{f_{h-1}(t_{h-1}) + q_{h-1,h}(t_{h-1}) \mid t_{h-1} \in D_l\}$), or to calculate the composite function (i.e., $\gamma_h(s) = f_{h-1}(t_{h-1}) + q_{h-1,h}(t_{h-1})$ where $t_{h-1} + \lambda_{h-1,h}(t_{h-1}) = s$ holds. We can compute t_{h-1} from s by taking the inverse of $t_{h-1} + \lambda_{h-1,h}(t_{h-1})$). In both cases, the time complexity is linear to the number of the corresponding linear pieces of f_{h-1}, $q_{h-1,h}$ and $\lambda_{h-1,h}$. During the whole computation of (i), we need to scan (the linked lists representing) the functions $\lambda_{h-1,h}(t_{h-1})$ and $f_{h-1}(t_{h-1}) + q_{h-1,h}(t_{h-1})$ only once from left to right. For completing γ_h by merging $\gamma_h|_{R_l}$ for $l = 1, 2, \ldots, L$ in (ii), it is straightforward to see that the time complexity is $O(L)$. Hence the time complexity of computing function $\gamma_h(s)$ is $O(\delta(f_{h-1}) + \delta(q_{h-1,h}) + \delta(\lambda_{h-1,h}))$, and $\delta(\gamma_h) \leq \delta(f_{h-1}) + \delta(q_{h-1,h}) + \delta(\lambda_{h-1,h})$ holds.

Now we can compute f_h by the recurrence formula (9) in $O(\delta(p_h) + \delta(\gamma_h))$ time, where the number of pieces of f_h is at most $\delta(p_h) + \delta(\gamma_h)$. Hence we can compute f_h from f_{h-1} in

$$\begin{aligned} &O\left(\delta(f_{h-1}) + \delta(q_{h-1,h}) + \delta(\lambda_{h-1,h})\right) + O\left(\delta(p_h) + \delta(\gamma_h)\right)) \\ &= O\left(\delta(f_{h-1}) + \delta(p_h) + \delta(q_{h-1,h}) + \delta(\lambda_{h-1,h})\right) \end{aligned}$$

time and we have

$$\begin{aligned} \delta(f_h) &\leq \delta(p_h) + \delta(\gamma_h) \\ &\leq \delta(f_{h-1}) + \delta(p_h) + \delta(q_{h-1,h}) + \delta(\lambda_{h-1,h}). \end{aligned}$$

Hence we have

$$\delta(f_h) \leq \sum_{h'=1}^{h} \delta(p_{h'}) + \delta(q_{h'-1,h'}) + \delta(\lambda_{h'-1,h'}).$$

Using this, the time complexity of computing f_h from f_{h-1} is evaluated as

$$O\left(\sum_{h'=1}^{h} \delta(p_{h'}) + \delta(q_{h'-1,h'}) + \delta(\lambda_{h'-1,h'})\right).$$

In summary, given a route σ_k, we can compute the forward minimum cost function of a customer from that of the previous customer in $O\left(\Delta(\sigma_k)\right)$ time, where

$$\Delta(\sigma_k) = \sum_{h=1}^{n_k+1} \delta(p_{\sigma_k(h)}) + \delta(q_{\sigma_k(h-1),\sigma_k(h)}) + \delta(\lambda_{\sigma_k(h-1),\sigma_k(h)}).$$

We can then obtain the optimal cost of σ_k in $O(n_k\Delta(\sigma_k))$ time by computing the forward minimum cost functions of n_k customers in σ_k, and taking the minimum of the forward minimum cost function of the depot. Note that $\Delta(\sigma_k)$ is the same as the input size of the OSTP. If traveling cost and time functions are constant functions (i.e., if there is no time dependency), this time complexity of the dynamic programming algorithm becomes the same as that of Ibaraki et al. [16].

3.3 Remarks for the case in which condition (1) does not hold

Even if condition (1) does not hold, we can compute $\gamma_h(s)$ in a similar manner as in Section 3.2. Figure 4 shows the same situation as Figure 3 in which condition (1) does not hold. In order to compute $\gamma_h(s)$, we split the domain of t_{h-1} into the intervals $D_1, D_2, \ldots, D_L$ as before, compute the functions

$$\tilde{\gamma}_h^l(s) = f_{h-1}(t_{h-1}) + q_{h-1,h}(t_{h-1}),$$

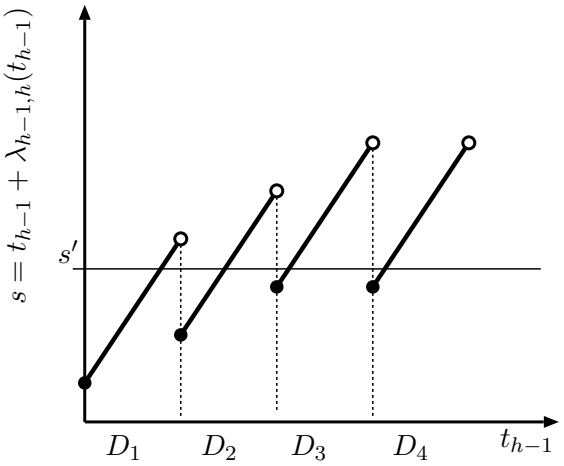


Figure 4: An example of λ_{ij} which does not satisfy condition (1)

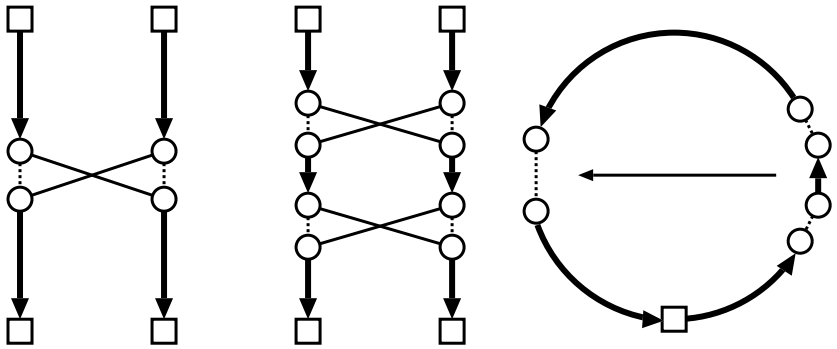
where $t_{h-1} + \lambda_{h-1,h}(t_{h-1}) = s$ and $t_{h-1} \in D_l$, for each R_l, $l = 1, 2, \ldots, L$, and complete γ_h by taking the lower envelope of them.

In general, the complexity of the lower envelope of n segments is $\theta(\alpha(n)n)$, where α denotes the inverse of Ackermann's function [17]. Using this fact, letting $\Delta = \delta(f_{h-1}) + \delta(q_{h-1,h}) + \delta(\lambda_{h-1,h})$, the complexity of γ_h is bounded by $O(\alpha(\Delta)\Delta)$. However, this upper bound is not small enough to prove that the complexity of f_h is of polynomial order. Hence our dynamic programming algorithm may require exponential time. Whether the algorithm runs in polynomial time or not, and whether the OSTP itself is NP-hard or not are both open.

4 Local search for finding visiting orders $\boldsymbol{\sigma}$

In this section, we describe the framework of our local search (LS) for finding good visiting orders $\boldsymbol{\sigma} = (\sigma_1, \sigma_2, \ldots, \sigma_m)$ that satisfy condition (3). It starts from an initial solution $\boldsymbol{\sigma}$ and repeats replacing $\boldsymbol{\sigma}$ with a better solution in its neighborhood $N(\boldsymbol{\sigma})$ until no better solution is found in $N(\boldsymbol{\sigma})$. As $N(\boldsymbol{\sigma})$ we use the standard neighborhoods called 2-opt*, cross exchange and Or-opt with slight modifications (see Figure 5). In this figure, squares represent the depot (which is duplicated at each end) and small circles represent customers in the routes. A thin line represents a route edge and a thick line represents a path (i.e., more than two customers may be included).

The 2-opt* neighborhood was proposed in [6], which is a variant of the 2-opt neighborhood [18] for the traveling salesman problem. A 2-opt* operation removes two edges from two different routes (one from each) to divide each route into two parts and exchanges the second parts of the two routes. The cross exchange neighborhood was proposed in [8]. A cross exchange operation removes two paths from two routes (one from each) of different vehicles, whose length (i.e., the number of customers in the path) is at most L^{cross} (a parameter), and exchanges them. The cross exchange and 2-opt* operations always change the assignment of customers to vehicles. We also use the intra-route neighborhood to improve individual routes, which is a variant of Or-opt neighborhood used for TSP [19, 20]. An intra-route operation removes a path of length at most $L^{\mathrm{intra}}_{\mathrm{path}}$ (a parameter) and inserts it into another position of the same route, where the position is limited within length $L^{\mathrm{intra}}_{\mathrm{ins}}$ (a parameter) from the original position. Our LS searches



(a) 2-opt* (b) Cross exchange (c) Or-opt

Figure 5: Neighborhoods in our local search

the above intra-route neighborhood, 2-opt* neighborhood and cross exchange neighborhood, in this order. Whenever a better solution is found, we immediately accept it (i.e., we adopt the first admissible move strategy), and resume the search from the intra-route neighborhood.

As only one execution of LS may not be sufficient to find a good solution, we use the *iterated local search* (ILS), which iterates LS many times from those initial solutions generated by perturbing good solutions obtained by then. We perturb a solution by applying one random cross exchange operation with no restriction on L^{cross} (i.e., $L^{\text{cross}} = n$). ILS is summarized as follows:

ILS

Step 1 Generate an initial solution $\boldsymbol{\sigma}^0$. Let $\boldsymbol{\sigma}^{\text{seed}} := \boldsymbol{\sigma}^0$ and $\boldsymbol{\sigma}^{\text{best}} := \boldsymbol{\sigma}^0$.

Step 2 Improve $\boldsymbol{\sigma}^{\text{seed}}$ by LS and let $\boldsymbol{\sigma}$ be the improved solution.

Step 3 If $\boldsymbol{\sigma}$ is better than $\boldsymbol{\sigma}^{\text{best}}$, then replace $\boldsymbol{\sigma}^{\text{best}}$ with $\boldsymbol{\sigma}$.

Step 4 If some stopping criterion is satisfied, output $\boldsymbol{\sigma}^{\text{best}}$ and halt; otherwise generate a solution $\boldsymbol{\sigma}^{\text{seed}}$ by perturbing $\boldsymbol{\sigma}^{\text{best}}$ and return to Step 2.

5 Efficient implementation of local search

A solution $\boldsymbol{\sigma}$ is evaluated by $(p+q)^*_{\text{sum}}(\boldsymbol{\sigma}) + a_{\text{sum}}(\boldsymbol{\sigma})$, where $(p+q)^*_{\text{sum}}(\boldsymbol{\sigma})$ denotes the minimum time window and traveling time cost for computed $\boldsymbol{\sigma}$ by dynamic programming in Section 3. (Actually in our algorithm, we split each traveling cost function into the constant part and the time-dependent part, and compute them separately to improve the efficiency.) For this, it is important to see that dynamic programming computation of $(p+q)^*_{\text{sum}}(\boldsymbol{\sigma})$ for the solutions in neighborhoods can be sped up by using information from the previous computation. Similar idea was originally proposed in Ibaraki et al. [16] for a simpler problem, in which the traveling times between customers were constants and no time-dependent traveling cost was considered. Below we will denote by $\langle \sigma_k(h_1) \to \sigma_k(h_2) \rangle$ the path from the h_1-th customer to the h_2-th customer in route σ_k, and by $\langle \sigma_k(h_1) \to \sigma_k(h_2) \rangle$–$\langle \sigma_{k'}(h_3) \to \sigma_{k'}(h_4) \rangle$ the path constructed by connecting two paths $\langle \sigma_k(h_1) \to \sigma_k(h_2) \rangle$ and $\langle \sigma_{k'}(h_3) \to \sigma_{k'}(h_4) \rangle$ from routes σ_k and $\sigma_{k'}$.

5.1 The basic idea

Consider the computation of the minimum cost (including the amount of capacity excess, the time window cost and the traveling time cost) for a given route $\sigma_k = (\sigma_k(0), \sigma_k(1), \ldots, \sigma_k(n_k{+}1))$ when it is obtained by connecting its former part $\langle 0 \to \sigma_k(h)\rangle$ and the latter part $\langle \sigma_k(h{+}1) \to 0\rangle$ for some h as illustrated in Figure 6. The amount of capacity excess for route σ_k is computed

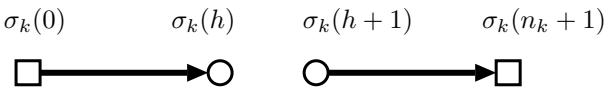


Figure 6: The former and latter parts of a route σ_k

in $O(1)$ time, if both $\sum_{i=1}^{h} a_{\sigma_k(i)}$ and $\sum_{i=h+1}^{n_k} a_{\sigma_k(i)}$ are known. We therefore store $\sum_{i=1}^{h} a_{\sigma_k(i)}$ and $\sum_{i=h}^{n_k} a_{\sigma_k(i)}$ for each customer $\sigma_k(h)$ and vehicle k whenever the current route is updated.

Now we concentrate on the computation of $(p+q)^*_{\mathrm{sum}}(\sigma_k)$, which is the minimum sum of time window and traveling time costs on route σ_k.

> Let $b^k_h(t)$ be the minimum sum of the time window costs for customers $\sigma_k(h), \sigma_k(h+1), \ldots, \sigma_k(n_k+1)$ and the traveling costs between them provided that all of them are served after time t.

We call this a *backward minimum cost function*. Then, $b^k_h(t)$ can be computed as follows, which is symmetric to the forward minimum cost computation discussed in Section 3:

$$\begin{aligned} b^k_{n_k+1}(t) &= \min_{s\geq t} p_0(s), \\ b^k_h(t) &= \min_{s\geq t}\{p_{\sigma_k(h)}(s) \\ &\quad + \min_{t'\geq s}\{b^k_{h+1}(t' + \lambda_{\sigma_k(h),\sigma_k(h+1)}(t')) + q_{\sigma_k(h),\sigma_k(h+1)}(t')\}\}, \\ &\qquad\qquad 1 \leq h \leq n_k. \end{aligned} \tag{11}$$

Let $f^k_h(t)$ be the forward minimum cost function at the hth customer in route σ_k. We can then obtain the optimal cost $(p+q)^*_{\mathrm{sum}}(\sigma_k)$ by

$$\min_t \left\{ b^k_h(t) + \min_{t'+\lambda_{\sigma_k(h-1),\sigma_k(h)}(t')\leq t} f^k_{h-1}(t') + q_{\sigma_k(h-1),\sigma_k(h)}(t') \right\} \tag{12}$$

for any h $(1 \leq h \leq n_k + 1)$. If $f^k_{h-1}(t)$ and $b^k_h(t)$ are already available for some h, this is possible in $O(\Delta(\sigma_k))$ time (since the computation of (12) is similar to that of (8)). To achieve this, we store all functions $f^k_h(t)$ and $b^k_h(t)$ for each customer $\sigma_k(h)$, when these were computed in the process of LS.

In summary, we can compute the minimum cost of route σ_k in $O(\Delta(\sigma_k))$ time, if we keep the data $\sum_{i=1}^{h} a_{\sigma_k(i)}$, $\sum_{i=h}^{n_k} a_{\sigma_k(i)}$, $f^k_h(t)$ and $b^k_h(t)$ for all $h = 1, 2, \ldots, n_k$ and $k \in M$.

In our algorithm, the number of pieces in forward and backward minimum cost functions is closely linked to the speed of our algorithm. Hence we consider its reduction. Since $f^k_h(t)$ (resp., $b^k_h(t)$) is nonincreasing (resp., nondecreasing), there are usually many pieces with considerably large values in $f^k_h(t)$ (resp., in $b^k_h(t)$) for small (resp., large) t. Such pieces will not be used in evaluating improved solutions. We therefore shrink those pieces whose minimum values over their intervals are larger than the objective value of the current solution, into one piece.

5.2 How to apply the basic idea to the solutions in neighborhoods

We now explain how to apply the above idea to evaluate solutions in the neighborhoods efficiently. We only discuss the sum of time window and traveling costs, since the amount of capacity excess can be similarly treated. Recall that we can compute the forward minimum cost function (resp., the backward minimum cost function) of a customer from that of the previous customer (resp., the next customer) in $O(\Delta(\sigma_k))$ time, and that we can evaluate the route cost by connecting the forward and backward minimum cost functions in $O(\Delta(\sigma_k))$ time.

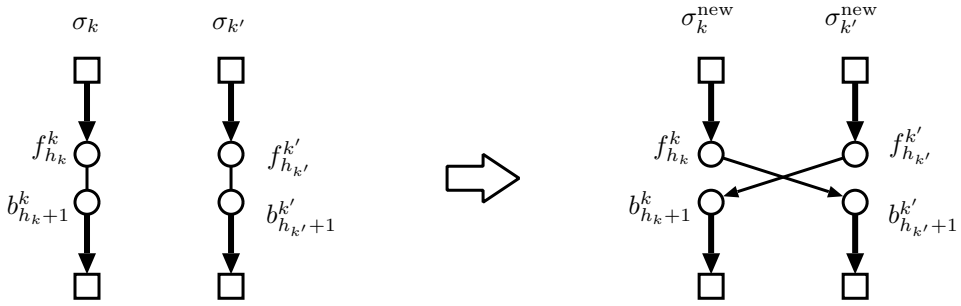


Figure 7: An example of a 2-opt* operation

In Figure 7, an example of a 2-opt* operation on routes σ_k and $\sigma_{k'}$ is shown. We denote by σ^{new}_k and $\sigma^{\mathrm{new}}_{k'}$ the resulting two routes (i.e., $\sigma^{\mathrm{new}}_k = \langle 0 \to \sigma_k(h_k)\rangle$–$\langle \sigma_{k'}(h_{k'}+1) \to 0\rangle$ and $\sigma^{\mathrm{new}}_{k'} = \langle 0 \to \sigma_{k'}(h_{k'})\rangle$–$\langle \sigma_k(h_k+1) \to 0\rangle$). Then, the sum of time window and traveling time costs for σ^{new}_k can be computed by

$$\min_t \left\{ b^{k'}_{h_{k'}+1}(t) + \min_{t'+\lambda_{\sigma_k(h_k),\sigma_{k'}(h_{k'}+1)}(t') \le t} f^k_{h_k}(t') + q_{\sigma_k(h_k),\sigma_{k'}(h_{k'}+1)}(t') \right\}$$

in $O\left(\Delta(\sigma^{\mathrm{new}}_k)\right)$ time. Similarly the cost for $\sigma^{\mathrm{new}}_{k'}$ can be computed in $O\left(\Delta(\sigma^{\mathrm{new}}_{k'})\right)$ time. Hence, when a 2-opt* operation is applied to routes σ_k and $\sigma_{k'}$, we can evaluate the cost of the resulting solution in $O\left(\Delta(\sigma^{\mathrm{new}}_k) + \Delta(\sigma^{\mathrm{new}}_{k'})\right)$ time.

To evaluate solutions in the cross exchange neighborhood efficiently, we need to search the solutions in the neighborhood in a specific order. To apply cross exchange operations on routes σ_k and $\sigma_{k'}$, we start from a solution obtainable by exchanging one customer from each route, and then extend lengths of the paths to be exchanged one by one. Figure 8 explains the situation. We denote by σ^{tmp}_k and $\sigma^{\mathrm{tmp}}_{k'}$ the routes obtained by applying a cross exchange operation on the current routes σ_k and $\sigma_{k'}$ (see Figure 8 (a)) and by σ^{new}_k and $\sigma^{\mathrm{new}}_{k'}$ the routes generated next (see Figure 8 (b)). In Figure 8 (a), backward minimum cost functions $b^k_{h_k}$, $b^{k'}_{h_{k'}}$ and $b^{k'}_{h_{k'}+1}$ of the current routes σ_k and $\sigma_{k'}$ are available, and we have already computed the forward minimum cost functions $\tilde{f}^k_1, \tilde{f}^k_2, \ldots, \tilde{f}^k_l$ (resp., $\tilde{f}^{k'}_1, \tilde{f}^{k'}_2, \ldots, \tilde{f}^{k'}_{l'}$) on the partial paths in σ^{tmp}_k (resp., $\sigma^{\mathrm{tmp}}_{k'}$) in the process of computing $(p+q)^*_{\mathrm{sum}}(\sigma^{\mathrm{tmp}}_k)$ (resp., $(p+q)^*_{\mathrm{sum}}(\sigma^{\mathrm{tmp}}_{k'})$). We then compute $\tilde{f}^k_{l+1}$ from $\tilde{f}^k_l$ by recursion of the dynamic programming in $O(\Delta(\sigma^{\mathrm{new}}_k))$ time, and evaluate $(p+q)^*_{\mathrm{sum}}(\sigma^{\mathrm{new}}_k) + (p+q)^*_{\mathrm{sum}}(\sigma^{\mathrm{new}}_{k'})$ in $O\left(\Delta(\sigma^{\mathrm{new}}_k) + \Delta(\sigma^{\mathrm{new}}_{k'})\right)$ time (Figure 8 (b)). Thus, we can compute the change in the cost after a cross exchange operation in $O\left(\Delta(\sigma^{\mathrm{new}}_k) + \Delta(\sigma^{\mathrm{new}}_{k'})\right)$ time.

Similarly, the change in the cost for an intra-route operation of route σ_k can be computed in $O\left(\Delta(\sigma^{\mathrm{new}}_k)\right)$ time, by searching solutions in a specific order, where σ^{new}_k denotes the route

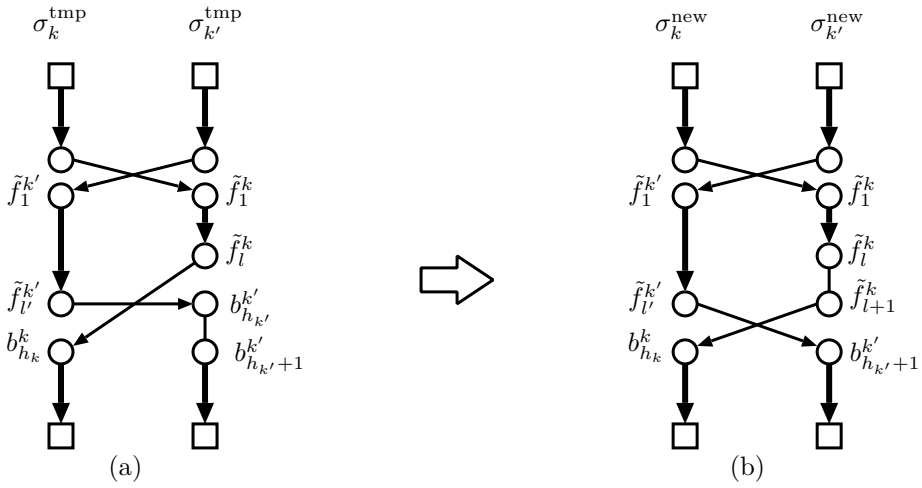


Figure 8: An example of the search order in the cross exchange neighborhood

generated by an intra-route operation. Actually, this case is slightly more complicated than the case of cross exchange neighborhood, but the search order described in Ibaraki et al. [16] also works for our problem.

5.3 Restriction of Neighborhoods

In searching neighborhoods, we find that there are many solutions which have no prospects of improvements. In order to avoid evaluating such solutions, we propose a rule to restrict the search.

For a constant U, let

$$F_h^k(U) = \begin{cases} \min\{t \mid f_h^k(t) \le U\}, & \text{if } \min_t f_h^k(t) \le U \\ +\infty, & \text{otherwise.} \end{cases}$$

This $F_h^k(U)$ gives the earliest departure time of vehicle k from customer $\sigma_k(h)$ in order to keep the sum of the time window cost of customers $\sigma_k(1)$, $\sigma_k(2), \ldots,$ $\sigma_k(h)$ and the traveling cost between them below U. In other words

$$t \ge F_h^k(U) \iff f_h^k(t) \le U$$

holds. As mentioned in Section 3.2, we store $f_h^k(t)$ in a linked list; however, we can also store $f_h^k(t)$ in an array without sacrificing the time complexity. Using this the array data structure, we can compute $F_h^k(U)$ for a given U in $O(\log(\delta(f_h^k)))$ time because f_h^k is a nonincreasing function. (In our program, however, we did not implement the array structure, and use $O(\delta(f_h^k))$ time to compute $F_h^k(U)$, because this does not seem to be a bottle neck of computation.)

Similarly let

$$B_h^k(U) = \begin{cases} \max\{t \mid b_h^k(t) \le U\}, & \text{if } \min_t b_h^k(t) \le U \\ -\infty, & \text{otherwise.} \end{cases}$$

$B_h^k(U)$ is the latest arrival time of vehicle k at customer $\sigma_k(h)$ in order to keep the time window cost of customers $\sigma_k(h), \sigma_k(h+1), \ldots, \sigma_k(n_k+1)$ and the traveling cost between them below U. Note also that

$$t \le B_h^k(U) \iff b_h^k(t) \le U$$

holds, because b_h^k is a nondecreasing function, and we can compute $B_h^k(U)$ in $O(\log(\delta(b_h^k)))$ time. Then, if

$$F_{h-1}^k(U) + \lambda_{k,h}^{\min} > B_h^k(U)$$

holds for $\lambda_{k,h}^{\min} = \min_t \lambda_{\sigma_k(h-1),\sigma_k(h)}(t)$, the cost of route σ_k must be larger than U. This fact is utilized to restrict the search in the 2-opt* and cross exchange neighborhoods, whose details are explained in Sections 5.3.1 and 5.3.2.

Furthermore, for any nonnegative nonincreasing function f and any nonnegative nondecreasing function b, we can obtain lower and upper bounds of $\min_t\{f(t)+b(t)\}$ by the following observation. Let a point $\hat{t}$ satisfy that $t \geq \hat{t} \Rightarrow b(t) \geq f(\hat{t})$ and $t \leq \hat{t} \Rightarrow f(t) \geq b(\hat{t})$, and we call this the *switch point* of f and b. Then the switch point $\hat{t}$ satisfies

$$\max\{f(\hat{t}), b(\hat{t})\} \leq \min_t\{f(t)+b(t)\} \leq f(\hat{t}) + b(\hat{t}) \leq 2\max\{f(\hat{t}), b(\hat{t})\}.$$

If there is no switch point, either $f(t) > b(t)$ or $f(t) < b(t)$ holds for all t. In this case if $f(t) > b(t)$ holds for all t, then

$$f(\tilde{t}) \leq \min_t\{f(t)+b(t)\} \leq f(\tilde{t}) + b(\tilde{t}) \leq 2f(\tilde{t})$$

holds, where $\tilde{t} = \arg\min_t f(t)$. When f and b are continuous, the switch point is the intersecting point of f and b.[1] From this property, for any $h \in \{1, 2, \ldots, n_k+1\}$, if $\lambda_{\sigma_k(h-1),\sigma_k(h)}(t)$ is a constant function and $q_{\sigma_k(h-1),\sigma_k(h)} = 0$, we can obtain a lower bound on the OSTP from the switch point of $f_{h-1}^k(t)$ and $b_h^k(t+\lambda_{\sigma_k(h-1),\sigma_k(h)}(t))$, and the schedule induced by the switch point becomes a 2-approximate schedule for σ_k (i.e., the cost of the schedule is at most $2(p+q)^*_{\mathrm{sum}}(\sigma_k)$), where cost can be computed in $O\left(\log(\delta(f_{h-1}^k)) + \log(\delta(b_h^k))\right) = O(\log(\Delta(\sigma_k)))$ time. Even if $\lambda_{\sigma_k(h-1),\sigma_k(h)}$ and $q_{\sigma_k(h-1),\sigma_k(h)}$ are time-dependent, the switch point of $f_{h-1}^k(t)$ and $b_h^k(t+\lambda_{k,h}^{\min})$ gives a lower bound on the optimal cost. Hence we can skip solving the OSTP optimally in the search of neighborhoods if its lower bound tells that it cannot improve the current $\boldsymbol{\sigma}$.

5.3.1 2-opt* neighborhood

Consider to evaluate a solution in the 2-opt* neighborhood obtained by reconnecting two routes σ_k and $\sigma_{k'}$ (see Figure 7). We set a threshold U, and avoid evaluating the rouets if we can conclude $(p+q)^*_{\mathrm{sum}}(\sigma_k^{\mathrm{new}}) + (p+q)^*_{\mathrm{sum}}(\sigma_{k'}^{\mathrm{new}}) > U$. As our purpose is to obtain a better solution than the current one, we can set U as the total cost of the current routes σ_k and $\sigma_{k'}$. Our first idea is based on the following fact: The solution obtained by connecting $\sigma_k(h_k)$ and $\sigma_{k'}(h_{k'}+1)$ will have a cost larger than U if $F_{h_k}^k(U) > B_{h_{k'}+1}^{k'}(U)$ holds. Let

$$P_{\mathrm{valid}}^{kk'}(U) = \left\{(\sigma_k(h_k), \sigma_{k'}(h_{k'})) \mid F_{h_k}^k(U) \leq B_{h_{k'}+1}^{k'}(U) \text{ and } F_{h_{k'}}^{k'}(U) \leq B_{h_k+1}^k(U)\right\}.$$

Then we can compute $P_{\mathrm{valid}}^{kk'}(U)$ in $O(n_k + n_{k'} + |P_{\mathrm{valid}}^{kk'}(U)|)$ time if $F_{h_k}^k(U)$, $B_{h_k}^k(U)$, $F_{h_{k'}}^{k'}(U)$ and $B_{h_{k'}}^{k'}(U)$ are available for all h_k and $h_{k'}$. It takes $O(n_k\Delta(\sigma_k) + n_{k'}\Delta(\sigma_{k'}))$ time for the preprocessing ($O(n_k \log(\Delta(\sigma_k)) + n_{k'} \log(\Delta(\sigma_{k'})))$ time if we use the array structure). Such preprocessing is necessary only when the current solution is changed (i.e., when an improved solution is found or a perturbation is applied), and is usually dominated by the evaluation time of solutions.

[1] In our implementation, we just took an intersecting point instead of a switch point, because all functions of the tested instances are continuous.

Let

$$P_{\mathrm{valid}}(U) = \bigcup_{k \neq k'} P^{kk'}_{\mathrm{valid}}(U).$$

Then we can compute $P_{\mathrm{valid}}(U)$ in $O(nm + |P_{\mathrm{valid}}(U)|)$ time. Any solution cannot be better than the current solution unless it is induced from $P_{\mathrm{valid}}(U)$.

Hence we restrict the 2-opt* neighborhood only to the solutions induced from $P_{\mathrm{valid}}(U)$. Although the size of the 2-opt* neighborhood is reduced from $O(n^2)$ to $O(|P_{\mathrm{valid}}(U)|)$ by this modification, we miss no better solution in the 2-opt* neighborhood.

5.3.2 Cross exchange neighborhood

Consider the search in the cross exchange neighborhood. The size of the cross exchange neighborhood is $O(n^2(L^{\mathrm{cross}})^2)$, and is largest in the standard neighborhoods used in this paper. Here we consider a restriction of the cross exchange neighborhood. In order to keep the change in the time window costs and traveling time costs small, it seems preferable to keep the arriving times at customers in the generated solution close to those of the current solution. Based on this intuition, we restrict the paths to be exchanged to those which satisfy $(\sigma_k(h_1^k), \sigma_{k'}(h_1^{k'})) \in P^{k,k'}_{\mathrm{valid}}(U)$, where $\sigma_k(h_1^k)$ and $\sigma_{k'}(h_1^{k'})$ are the first customers of the paths. Note that, different from the case of 2-opt* neighborhood, this restriction has a possibility of missing a better solution in the cross exchange neighborhood.

6 Computational results

We conducted computational experiments to evaluate the proposed algorithm ILS. The algorithm was coded in C language and run on a handmade PC (Intel Pentium 4, 2.8 GHz, 1 GB memory). We used $L^{\mathrm{cross}} = 3$, $L^{\mathrm{intra}}_{\mathrm{path}} = 3$ and $L^{\mathrm{intra}}_{\mathrm{ins}} = 15$ in the experiments.

6.1 Effect of the restriction of the neighborhoods

We first consider the effect of the restriction of the neighborhoods discussed in Section 5.3. We run our local search algorithm with the 2-opt* neighborhood only, from a random solution and a locally optimum solution, both with and without restriction. We use the same random solution and the same locally optimal solution for the initial solutions of the runs with and without restriction. In a similar manner, we also test our local search algorithm with the cross exchange neighborhood only. For these tests, we used the instance r201 in Solomon's benchmark list, whose details will be described in Section 6.2.

Figures 9 shows the results with the 2-opt* neighborhood without (left) and with (right) restriction, respectively, whose the vertical axis gives the total cost of every two routes constructed as neighborhood solutions, while the horizontal axis shows the cost before the neighborhood operation is applied (i.e., the current solution). Namely, each point in the figure corresponds to the two route cost of a neighborhood solution. Similarly, Figure 10 shows the results with the cross exchange neighborhood.

From these figures, we observe that the proposed restriction succeeds in avoiding the evaluations of solutions whose costs are much larger than that of the current solutions. We can also observe that the restriction becomes more effective when the cost of the current solution is small.

Then Table 1 shows the number of cost evaluations needed to obtain a locally optimal solution, with and without restriction of neighborhood. Column "random" (resp., "locally optimal")

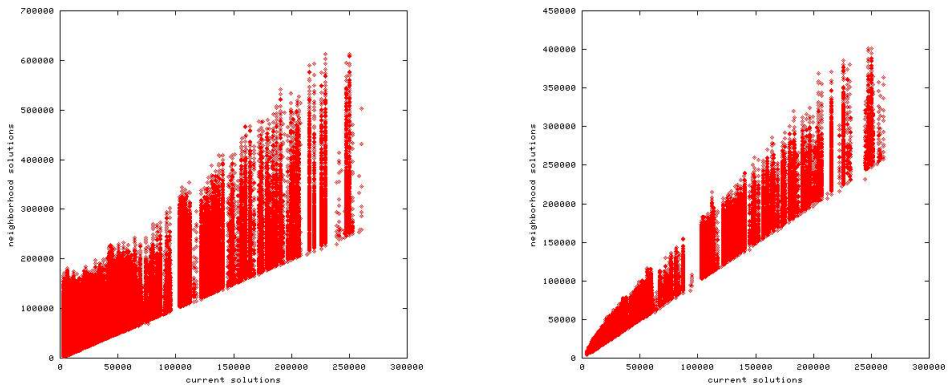


Figure 9: The distribution of two route cost in the 2-opt* neighborhood without (left) and with (right) restriction

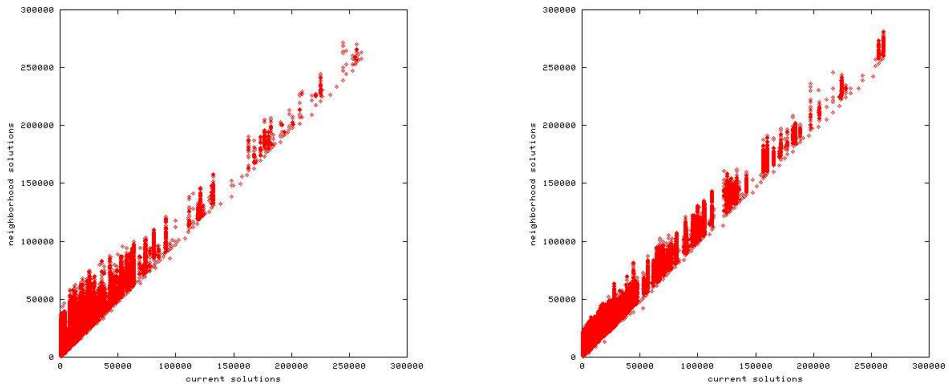


Figure 10: The distribution of two route cost in the cross exchange neighborhood without (left) and with (right) restriction

Table 1: Number of evaluations with and without restriction of neighborhood

neighborhood	2-opt*		cross exchange	
initial solution	random	locally optimal	random	locally optimal
without restriction	227746(11016.59)	4035(1253.23)	333321 (1607.82)	24875(1253.23)
with restriction	146440(17906.10)	164(1253.23)	139771 (1518.22)	111(1253.23)

shows the number of evaluations during the local search when the initial solution is a random (resp., locally optimal) solution. Note that, in the case of the locally optimal initial solution, no improvement is achieved as a result of local search. The two rows correspond to the cases with and without restriction, respectively, where the values in parenthesis are the objective values of the obtained locally optimal solutions.

From Table 1, we can confirm the effectiveness of the restriction. In the neighborhood of a random solution we can reduce the number of evaluations to almost a half, and in the neighborhood of a locally optimal solution, we can reduce it to only a few percent. In our restriction of 2-opt* neighborhood, the solution quality is basically the same since the restriction is guaranteed not to miss any improved neighborhood solution. However, the objective values are different in the table, because we use random numbers when we determine the search order in neighborhood. Although we may miss improved solutions in the case of the cross exchange neighborhood with restriction, the output solution happens to be slightly better than that obtained without restriction in this particular case. Since each run of local search resumes from a solution generated by applying a small perturbation on a good locally optimal solution in our ILS algorithm, the effect of the restriction is expected to be significant.

6.2 The vehicle routing problem with hard time windows

We used Solomon's benchmark instances [21] and Gehring and Homberger's benchmark instances [22], which have been widely used in the literature. We first explain Solomon's instances. The number of customers in each instance is 100, and their locations are distributed in the square $[0, 100]^2$ in the plane. The distances between customers are measured by Euclidean distances (in double precision), and the traveling times are the same as the corresponding distances. Each customer i (including the depot) has one time window $[r_i, d_i]$, an amount of requirement a_i and a service time b_i. All vehicles have an identical capacity u. Both time window and capacity constraints are considered hard. For these instances, the number of vehicles m is also a decision variable, and the objective is to find a solution with the minimum vehicle number and the total traveling distance in the lexicographical order. These benchmark instances consist of six different sets of problem instances called R1, R2, RC1, RC2, C1 and C2, respectively. Locations of customers are uniformly and randomly distributed in type R and are clustered into groups in type C, and these two types are mixed in type RC. Furthermore, for the instances of type 1, the time window is narrow at the depot, and hence only a small number of customers can be served by one vehicle. On the contrary, for the instances of type 2, the time window at the depot is wide, and many customers can be served by one vehicle. Recently, 300 instances with larger number of customers are added by Gehring and Homberger [22], which are divided into five groups by the number of customers, 200, 400, 600, 800 and 1000, where each group has 10 instances for each of six types (i.e., R1, R2, RC1, RC2, C1 and C2).

In order to handle the above instances by our algorithm, we define time window cost function

p_i, traveling cost functions q_{ij} and traveling time functions λ_{ij} as follows:

$$\begin{aligned} p_i(t) &= \begin{cases} \alpha(r_i - t), & t < r_i \\ 0, & r_i \le t \le d_i \\ \alpha(t - d_i), & d_i < t, \end{cases} \\ q_{ij}(t) &= c_{ij}, \\ \lambda_{ij}(t) &= b_i + c_{ij}, \end{aligned}$$

where α is a positive parameter and c_{ij} is the distance (as well as the traveling time) between customers i and j. Note that, in this formulation, the time window constraint is considered as soft, and can be violated if it is advantageous from the view point of minimizing the cost function. We set the number of vehicles in each instance to what is used in [23].

We first conduct preliminary experiments to determine parameter value α for each instance. We run the algorithm with some values of α from $\{1, 5, 10, 50, 100, 500, \ldots\}$ in the manner of binary search, where the time limit for each α was within 10% of the time limit reported in the tables in this section. Then we use the best α among them and the adjacent values in both directions (e.g., if the best results was obtained with $\alpha = 50$, we use 10, 50 and 100 for α), run the algorithm by using the three values of α with 100% of the time limit, and report the best result below. If we cannot find a feasible solution, which satisfies the hard time window and capacity constraints, even with $\alpha = 1000000$, we increase the number of vehicles by one. Actually we could find a feasible solution for every instance except for six instances, and, for the six instances, we could find feasible solutions with one more vehicle.

We then compare the solutions obtained by our experiments with those obtained by existing methods. For Solomon's instances, the time limit of our algorithm is 1000 seconds for each instance. The results are shown in Table 2. In this table, "CNV" represents the cumulative number of vehicles, and "CTD" represents the cumulative total distance, which are usually used in the literature to compare the results on Solomon's instances. The upper (resp., lower) part of each cell in the table shows the mean number of vehicles (resp., the mean total distance) with respect to all instances for the type. Columns "IIKMUY", "HG99", "GH02", "BBB", "B", "BVH", "HG03", "IINSUY" and "ILS" are the results of Ibaraki et al. [16], Homberger and Gehring [24], Gehring and Homberger [25], Berger et al. [26], Bräysy [27], Bent and Van Hentenryck [28], Homberger and Gehring [22], Ibaraki et al. [23] and our ILS algorithm, respectively. The bottom rows describe the computer, the average CPU time and the number of independent runs for each method reported by the author, where "P" and "SU" mean Pentium and SUN Ultra, respectively. The row "Computer" contains the clock frequency of the computer (e.g., "P 200" means a computer whose CPU is Pentium 200MHz). Computation time of "HG03" is not clearly stated in [22]. To make a fair comparison of the performance of various algorithms, we estimate the total computation time for each experiment by using the SPEC data presented in the web page of SPEC (http://www.specbench.org/). The row "Estimated time" represents this estimated time. An asterisk "$*$" in rows of the mean number of vehicles indicates that the value is the best among all the algorithms in the table and there is no tie. When there are ties for the number of vehicles, we give an asterisk "$*$" on the corresponding distance value that is the smallest among those ties. In the row CNV, all results with the smallest value get "$*$".

The results for Gehring and Homberger's instances are given in Tables 3–7. The time limit of our algorithm for 200, 400, 600, 800 and 1000-customer instances are 2000, 4000, 6000, 8000 and 10000 seconds, respectively. Columns "GH99", "GH01", "BVH", "LL", "LC","BHD" "MB" and "IINSUY" are the results by Gehring and Homberger [29], Gehring and Homberger [25], Bent and Van Hentenryck [28], Li and Lim [30], Le Bouthillier and Crainic [31], Bräysy et al. [32],

Mester and Bräysy [33] and Ibaraki et al. [23] respectively. "AMD" in the row "Computer" means Advanced Micro Devices and "n/a" in the row "CPU(min)" and "Runs" means that the data is not available.

In Table 2, our ILS obtained CNV 405 and the smallest CTD among all the tested algorithms. According to a recent survey by Bräysy and Gendreau [10], not many algorithms achieved CNV 407 or less, and only those algorithms cited in Table 2 achieved CNV 405. In Tables 3–7, we could also obtain the smallest CNV among the tested algorithms. The computation time of our ILS is reasonable compared to others especially for larger instances. These results indicate that our ILS is highly efficient to solve the vehicle routing problem with time windows, in spite of its high generality. More details of the computational results for those instances to which we found better solutions than the best known solutions are presented in Tables 8–12. Column "Inst" shows the names of the instances, column "distance" shows the total traveling distances and column "bknown" shows the best known solutions for these instances, where the data was taken as of June 2, 2004 from

http://www.sintef.no/static/am/opti/projects/top/vrp/benchmarks.html.

An asterisk "$*$" in a row means that the value is better than that of "bknown". The details of the computational results for all instances are contained in Hashimoto [34]. Here it should be noted that we had to determine parameter α by preliminary experiments to achieve the above results, since the performance is crucially dependent on the value. Though the time spent for such tuning was not so large, it is one of the important directions of our future research to develop a mechanism to find a good value of α automatically.

Table 2: The results for 100-customer benchmark instances

	IIKMUY	HG99	GH02	BBB	B	BVH	HG03	IINSUY	ILS
C1	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00	10.00
	828.38*	828.38*	828.63	828.48	828.38*	828.38*	828.38*	828.38*	828.38*
C2	3.00	3.00	3.00	3.00	3.00	3.00	3.00	3.00	3.00
	589.86*	589.86*	590.33	589.93	589.86*	589.86*	589.86*	589.86*	589.86*
R1	11.92	11.92	12.00	11.92	11.92	11.92	11.92	12.00	11.92
	1217.40	1228.06	1217.57	1221.10	1222.12	1213.25	1212.73*	1217.99	1213.18
R2	2.73	2.73	2.73	2.73	2.73	2.73	2.73	2.73	2.73
	959.11	969.95	961.29	975.43	975.12	966.37	955.03*	967.97	955.61
RC1	11.50	11.63	11.50	11.50	11.50	11.50	11.50	11.63	11.50
	1391.03	1392.57	1395.13	1389.89	1389.58	1384.22*	1386.44	1384.67	1384.25
RC2	3.25	3.25	3.25	3.25	3.25	3.25	3.25	3.25	3.25
	1122.79	1144.43	1139.37	1159.37	1128.38	1141.24	1123.17	1128.77	1120.50*
CNV	405*	406	406	405*	405*	405*	405*	407	405*
CTD	57444	57876	57641	57952	57710	57567	57309	57545	57282*
Computer	P 1GHz	P 200	P400	P 400	P 200	SU 10	unknown	P 2.8GHz	P2.8GHz
CPU (min)	250.0	13.8	4×20.9	30.0	87.0	120.0	n/a	16.7	16.7
Runs	1	10	5	3	1	5	n/a	1	3
Estimated time	108.7	6.0	43.6	9.4	3.8	104.3	n/a	16.7	16.7

6.3 Time-dependent VRPSTW

Our algorithm ILS is designed for more general problem than the standard VRPSTW, i.e., time-dependent VRPSTW. In order to test the performance of ILS, we generated 56 instances

Table 3: The results for 200-customer benchmark instances

	GH99	GH01	BVH	LL	LC	BHD	MB	IINSUY	ILS
C1	18.9	18.9	18.9	19.1	18.9	18.9	18.8*	18.9	18.9
	2782	2842.08	2726.63	2728.6	2743.66	2749.83	2717.21	2732.03	2721.94
C2	6	6	6	6	6	6	6	6	6
	1846	1856.99	1860.17	1854.9	1836.1	1842.65	1833.57*	1834.83	1835.96
R1	18.2	18.2	18.2	18.3	18.2	18.2	18.2	18.2	18.2
	3705	3855.03	3677.96	3736.2	3676.95	3718.3	3618.68*	3665.77	3690.34
R2	4	4	4.1	4.1	4	4	4	4	4
	3055	3032.49	3023.62	3023	2986.01	3014.28	2942.92*	2965.64	2943.88
RC1	18	18.1	18	18.3	18	18	18	18	18
	3555	3674.91	3279.99	3385.8	3449.71	3329.62	3221.34*	3287.61	3345.01
RC2	4.3	4.4	4.5	4.9	4.3	4.4	4.4	4.3	4.3
	2675	2671.34	2603.08	2518.7	2613.75	2585.89	2519.79	2562.56*	2564.68
CNV	694*	696	697	707	694*	695	694*	694*	694*
CTD	176180	179328	171715	172472	173061	172406	168573*	170484	171018
Computer	P 200	P 400	SU 10	P 545	P 933	AMD 700	P 2GHz	P 2.8GHz	P 2.8GHz
CPU (min)	4×10	4×2.1	n/a	182.1	5×10	2.4	8	33.3	33.3
Runs	1	3	n/a	3	1	3	1	1	3
Estimated time	2.4	3.8	n/a	112.4	21.7	2.1	5.9	33.0	33.0

Table 4: The results for 400-customer benchmark instances

	GH99	GH01	BVH	LL	LC	BHD	MB	IINSUY	ILS
C1	38	38	38	38.7	37.9	37.9	37.9	37.7	37.6*
	7584	7855.82	7220.96	7181.4	7447.09	7230.48	7148.27	7282.15	7444.06
C2	12	12	12	12.1	12	12	12	12	11.8*
	3935	3940.19	4154.4	4017.1	3940.87	3894.48	3840.85	3851.96	3982.50
R1	36.4	36.4	36.4	36.6	36.5	36.4	36.3*	36.4	36.4
	8925	9478.22	8713.37	8912.4	8839.28	8692.17	8530.03	8746.94	8998.63
R2	8	8	8	8	8	8	8	8	8
	6502	6650.28	6959.75	6610.6	6437.68	6382.63	6209.94*	6269.9	6258.00
RC1	36.1	36.1	36.1	36.5	36	36	36	36	36
	8763	9294.99	8330.98	8377.9	8652.01	8305.55	8066.44*	8405.32	8572.11
RC2	8.6	8.8	8.9	9.5	8.6	8.9	8.8	8.6	8.5*
	5518	5629.43	5631.7	5466.2	5511.22	5407.87	5243.06	5337.5	5355.59
CNV	1390	1392	1393	1414	1390	1391	1389	1387	1383*
CTD	412270	428489	410112	405656	408281	399132	390386	398938	406109
Computer	P 200	P 400	SU 10	P 545	P 933	AMD 700	P 2GHz	P 2.8GHz	P 2.8GHz
CPU (min)	4×20	4×7.1	n/a	359.8	5×20	7.9	17	66.6	66.6
Runs	1	3	n/a	3	1	3	1	1	3
Estimated time	4.8	13.0	n/a	221.8	43.3	6.8	12.5	66.6	66.6

Table 5: The results for 600-customer benchmark instances

	GH99	GH01	BVH	LL	LC	BHD	MB	IINSUY	ILS
C1	57.9	57.7	57.8	58.2	57.9	57.8	57.8	57.5	57.5
	14792	14817.25	14357.11	14267.30	14205.58	14165.90	14003.09	14116.97*	14296.96
C2	17.9	17.8	17.8	18.2	17.9	18	17.8	17.4	17.4
	7787	7889.96	8259.04	8202.60	7743.92	7528.73	7455.83	7945.56*	7960.138
R1	54.5	54.5	55	55.2	54.8	54.5	54.5	54.5	54.5
	20854	21864.47	19308.62	19744.80	19869.82	19081.18	18358.68*	19844.39	20363.15
R2	11	11	11	11.1	11.2	11	11	11	11
	13335	13656.15	14855.43	13592.40	13093.97	13054.83	12703.52	12539.78*	13047.18
RC1	55.1	55	55.1	55.5	55.2	55	55	55	55
	18411	19114.02	17035.91	17320.00	17678.13	16994.22	16418.63*	17278.81	17764.33
RC2	11.8	11.9	12.4	13	11.8	12.1	12.1	11.6	11.5*
	11522	11670.29	11987.89	11204.90	11034.71	11212.36	10677.46	10791.70	11315.28
CNV	2082	2079	2091	2112	2088	2084	2082	2070	2069*
CTD	867010	890121	858040	843320	836261	820372	796172	825172	847470
Computer	P 200	P 400	SU 10	P 545	P 933	AMD 700	P 2GHz	P 2.8GHz	P 2.8GHz
CPU (min)	4×30	4 × 12.9	n/a	399.8	5× 30	16.2	40	100	100
Runs	1	3	n/a	3	1	3	1	1	3
Estimated time	7.1	23.5	n/a	246.4	65.0	13.9	29.4	100.0	100.0

Table 6: The results for 800-customer benchmark instances

	GH99	GH01	BVH	LL	LC	BHD	MB	IINSUY	ILS
C1	76.7	76.1	76.1	77.4	76.3	76.3	76.2	75.7	75.6*
	26528	26936.68	25391.67	25337.02	25668.82	25170.88	25132.27	25487.55	25915.59
C2	24	23.7	24.4	24.4	24.1	24.2	23.7	23.4	23.4
	12451	11847.92	14253.83	11956.60	11985.11	11648.92	11352.29	11860.90*	11942.54
R1	72.8	72.8	72.7*	73	73.1	72.8	72.8	72.8	72.8
	34586	34653.88	33337.91	33806.34	33552.40	32748.06	31918.47	33275.72	34095.04
R2	15	15	15	15.1	15	15	15	15	15
	21697	21672.85	24554.63	21709.39	21157.56	21170.15	20295.28	20209.92*	20810.51
RC1	72.4	72.3	73	73.2	72.3	73	73	72.4	72.3
	38509	40532.35	30500.15	31282.54	37722.62	30005.95	30731.07	34621.63	34358.45*
RC2	16.1	16.1	16.6	17.1	15.8	16.3	15.8	15.7	15.6*
	17741	17941.23	18940.84	17561.22	17441.60	17686.65	16729.18	16666.76	17173.59
CNV	2770	2760	2778	2802	2766	2776	2765	2750	2747*
CTD	1515120	1535849	1469790	1416531	1475281	1384306	1361586	1421225	1442957
Computer	P 200	P 400	SU 10	P 545	P 933	AMD 700	P 2GHz	P 2.8GHz	P 2.8GHz
CPU (min)	4×40	4×23.2	n/a	512.9	5×40	26.2	145	133.3	133.3
Runs	1	3	n/a	3	1	3	1	1	3
Estimated time	9.5	42.3	n/a	316.1	86.6	22.5	106.5	133.3	133.3

Table 7: The results for 1000-customer benchmark instances

	GH99	GH01	BVH	LL	LC	BHD	MB	IINSUY	ILS
C1	96	95.4	95.1	96.3	95.3	95.8	95.1	94.5	94.4*
	43273	43392.59	42505.35	42428.50	43283.92	42086.77	41569.67	42459.35	43066.89
C2	30.2	29.7	30.3	30.8	29.9	30.6	29.7	29.4	29.4
	17570	17574.72	18546.13	17294.90	17443.50	17035.88	16639.54	16986.46	16822.82*
R1	91.9	91.9	92.8	92.7	92.2	92.1	92.1	91.9	91.9
	57186	58069.61	51193.47	50990.80	55176.95	50025.64	49281.48	53366.10*	54149.50
R2	19	19	19	19	19.2	19	19	19	19
	31930	31873.62	36736.97	31990.90	30919.77	31458.23	29860.32	29546.19*	30626.04
RC1	90	90.1	90.2	90.4	90	90	90	90	90
	50668	50950.14	48634.15	48892.40	49711.36	46736.92	45396.41*	48275.20	49378.71
RC2	19	18.5	19.4	19.8	18.5	19	18.7	18.3	18.3
	27012	27175.98	29079.78	26042.30	26001.11	25994.12	25063.51	24904.08*	26428.81
CNV	3461	3446	3468	3490	3451	3465	3446	3431	3430*
CTD	2276390	2290367	2266959	2176398	2225366	2133376	2078110	2155374	2204728
Computer	P 200	P 400	SU 10	P 545	P 933	AMD 700	P 2GHz	P 2.8GHz	P 2.8GHz
CPU (min)	4×50	4×30.1	n/a	606.3	5×50	39.6	600	166.7	166.7
Runs	1	3	n/a	3	1	3	1	1	3
Estimated time	11.9	54.9	n/a	373.5	108.3	34.0	440.8	166.7	166.7

Table 8: Detailed results for 200-customer benchmark instances

				bknown	
Inst	m	α	distance	m	distance
c103	18	1	2707.35*	18	2708.08
c110	18	5	2647.92*	18	2649.26
c208	6	1	1820.53*	6	1820.59
c210	6	1	1806.58*	6	1806.60
r203	4	1000	2891.23*	4	2932.44
r205	4	5	3367.53*	4	3367.55
rc206	4	100	2889.42*	4	3138.02

Table 9: Detailed results for 400-customer benchmark instances

				bknown	
Inst	m	α	distance	m	distance
c102	36*	5000	7856.66	37	7357.45
c107	39	5000	7505.24*	39	8043.18
c108	37*	10000	7882.36	38	7113.40
c204	11*	1000000	4350.20	12	3535.99
c205	12	1	3938.69*	12	3939.42
c210	11*	5000	4257.64	12	3684.89
r202	8	1000	7662.25*	8	7674.90
r209	8	5	6486.50*	8	6493.13
rc202	9*	10000	6419.16	10	5924.84
rc203	8	5	5048.38*	8	5114.76
rc205	9	50	6047.21*	9	6063.46
rc206	8	50	5998.19*	8	6054.21

Table 10: Detailed results for 600-customer benchmark instances

				bknown	
Inst	m	α	distance	m	distance
c102	56	500	14209.47*	56	14325.96
c106	60	50	14089.66*	60	14089.70
c107	59	10000	14580.31*	59	14659.74
c108	56*	10000	15437.77	57	14976.88
c202	17*	1000	8784.11	18	7486.88
c203	17	50	7977.15*	17	8371.07
c207	18	10	7535.05*	18	7560.53
c208	17*	100	8169.53	18	7352.42
c209	17*	1000	9168.68	18	7350.94
rc201	14*	100	13753.35	15	13275.93
rc205	12*	10000	12757.40	13	11919.72
rc206	11*	1000	13396.83	12	11411.08

Table 11: Detailed results for 800-customer benchmark instances

				bknown	
Inst	m	α	distance	m	distance
c102	74*	1000000	26114.66	75	25518.17
c202	23*	10000	12773.63	24	11422.34
c105	80	100	25166.28*	80	25166.30
c106	80	100	25160.85*	80	25160.90
c206	24	1	11348.43*	24	11357.86
c108	75*	10000	26243.46	76	25379.85
c208	23*	10000	12195.91	24	11206.32
c209	23*	10000	13069.53	24	11249.00
rc201	19*	10000	20716.21	20	19989.12
rc102	72	50	35112.82*	72	39696.20
rc202	16*	10000	19129.08	17	18099.68
rc103	72	100	33015.08*	72	35577.87
rc104	72	10	30085.34*	72	32654.10
rc107	72	500	39643.15*	72	43829.43
rc108	72	100	36512.04*	72	43694.60
rc109	72	50	35660.83*	72	41816.70
rc110	72	10	34716.75*	72	41182.44

Table 12: Detailed results for 1000-customer benchmark instances

				bknown	
Inst	m	α	distance	m	distance
c102	90*	1000	45854.82	92	42920.70
c105	100	100	42469.18*	100	42469.20
c106	100	100	42471.28*	100	42471.30
c108	94*	10000	43555.1	96	42170.31
c109	91	1000	42755.59*	91	45386.93
c204	29	5	16896.99*	29	17153.19
c205	30	5	16568.73*	30	16586.46
c206	30	5	16348.20*	30	16371.65
c207	30*	10000	16827.81	31	16578.42
c208	29	1	16532.88*	29	18662.10
c209	29*	10000	17462.68	30	16651.96
r105	91	50	61437.30*	91	70838.01
rc201	21*	100	30585.71	22	30320.41
rc202	18*	1000000	29525.90	19	26592.40

of time-dependent VRPSTW by modifying Solomon's instances, as suggested by Ichoua et al. [15].

In these instances, all traveling between customers are categorized into three types, and the scheduling horizon (i.e., the time window at the depot) consists of morning, daytime and evening. The travel speed of a vehicle depends on the category and the period of scheduling horizon, which is further classified into three scenarios as shown in Table 13. Time dependency is small, medium and large in scenarios 1, 2 and 3, respectively. Note that the average speed in each scenario is approximately 1, and the difficulty of time windows constraints is similar to Solomon's instances. We define the time window cost function as

$$p_i(t) = \begin{cases} r_i - t, & t < r_i \\ 0, & r_i \le t \le d_i \\ t - d_i, & d_i < t, \end{cases}$$

and we construct each traveling time function $\lambda_{ij}(t)$ from the distance and travel speed in Table 13. We then define $q_{ij}(t) = \lambda_{ij}(t)$.

Table 13: Travel speed matrices for scenarios 1–3

	scenario1			scenario2			scenario3		
	morning	daytime	evening	morning	daytime	evening	morning	daytime	evening
category 1	0.54	0.81	0.54	0.33	0.67	0.33	0.12	0.46	0.12
category 2	0.81	1.22	0.81	0.67	1.33	0.67	0.46	1.92	0.46
category 3	1.22	1.82	1.22	1.33	2.67	1.33	0.96	3.84	0.96

Table 14 shows the computational results of our algorithm ILS for these instances. Column "time-dependent" represents the results of our algorithm devised for the time-dependent instances. Column "const" represents the results obtained by the following method, which was tested for comparison purposes: We solved the instances with our algorithm after replacing the time-dependent traveling time with the fixed constant determined by taking the average of the traveling time in the whole periods. Note that, in the case of "const", even though the constant

traveling times were used during the search, the final costs output by this method were evaluated exactly under the time-dependent environment, and the table shows the results under the exact evaluation. Each row gives the instance type, and the average values of p_{sum} and q_{sum} with respect to all instances of the same type. We omitted a_{sum}, since a_{sum} were always 0.

In Table 14, we can observe that both p_{sum} and q_{sum} in column "time-dependent" are smaller than those in column "const". The deference becomes larger as the instances become more time-dependent (i.e., from scenarios 1 to 3). This indicates the usefulness of our algorithm that can accept time-dependency.

Table 14: The results for time-dependent VRPSTW

	scenario1				scenario2				scenario3			
	time-dependent		const		time-dependent		const		time-dependent		const	
	p_{sum}	q_{sum}	p_{sum}	q_{sum}	p_{sum}	q_{sum}	p_{sum}	q_{sum}	p_{sum}	q_{sum}	p_{sum}	q_{sum}
C1	20.35	855.27	35.47	984.26	90.80	885.66	183.30	1301.37	342.45	1137.45	1312.33	2356.09
C2	2.31	669.94	0.00	712.52	7.43	763.61	54.25	919.15	325.91	1038.11	1554.59	1680.48
R1	32.47	1061.93	50.60	1188.67	43.41	921.41	94.92	1260.24	109.90	946.06	462.54	1378.72
R2	5.04	904.40	19.62	1021.33	3.00	809.01	77.37	1139.08	17.70	873.04	954.12	1562.06
RC1	50.10	1103.62	59.50	1298.58	55.23	967.13	96.89	1295.63	90.90	1023.89	486.02	1568.13
RC2	19.69	976.53	25.09	1140.38	12.40	877.58	106.79	1263.43	49.99	948.67	803.88	1640.06

7 Conclusion

We generalized the standard vehicle routing problem with time windows by allowing both traveling times and traveling costs to be time-dependent functions, and proposed an iterated local search algorithm. Our generalization can treat time-dependent traveling times and costs such as rush-hour traffic jam, and includes various interesting problems such as parallel machine scheduling problems as its special cases.

In our local search procedure, for each vehicle route generated during the search, we must compute an optimal schedule for the route. We showed that this subproblem can be efficiently solved by dynamic programming. We further proposed a filtering method that restricts the size of neighborhoods, based on the fact that there are many solutions having no prospect of improvement. We developed an iterated local search algorithm incorporating all the above ingredients. The computational results on representative benchmark instances indicate that the proposed algorithm is highly efficient. Artificially generated instances of the time-dependent vehicle routing problem with time windows were also solved to show the usefulness of our algorithm having high generality.

References

[1] M. Desrochers, J. K. Lenstra, M. W. P. Savelsbergh, F. Soumis, Vehicle routing with time windows: Optimization and approximation, in: B. L. Golden, A. A. Assad (Eds.), Vehicle Routing: Methods and Studies, North-Holland, Amsterdam, 1988, pp. 65–84.

[2] J. Desrosiers, Y. Dumas, M. M. Solomon, F. Soumis, Time constrained routing and scheduling, in: M. O. Ball, T. L. Magnanti, C. L. Monma, G. L. Nemhauser (Eds.), Handbooks in

Operations Research and Management Science, Vol. 8 of Network Routing, North-Holland, Amsterdam, 1995, pp. 35–139.

[3] M. M. Solomon, J. Desrosiers, Time window constrained routing and scheduling problems, Transportation Science 22 (1988) 1–13.

[4] G. Dantzig, R. Fulkerson, S. Johnson, Solution of a large-scale traveling-salesman problem, Journal of the Operations Research Society of America 2 (1954) 393–410.

[5] Y. A. Koskosidis, W. B. Powell, M. M. Solomon, An optimization-based heuristic for vehicle routing and scheduling with soft time window constraints, Transportation Science 26 (1992) 69–85.

[6] J. Y. Potvin, T. Kervahut, B. L. Garcia, J. M. Rousseau, The vehicle routing problem with time windows, part I: Tabu search, INFORMS Journal on Computing 8 (1996) 158–164.

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

[8] E. Taillard, P. Badeau, M. Gendreau, F. Guertin, J. Y. Potvin, A tabu search heuristic for the vehicle routing problem with soft time windows, Transportation Science 31 (1997) 170–186.

[9] O. Bräysy, M. Gendreau, Vehicle routing problem with time windows, part I: Route construction and local search algorithms, Transportation Science 39 (2005) 104–118.

[10] O. Bräysy, M. Gendreau, Vehicle routing problem with time windows, part II: Metaheuristics, Transportation Science 39 (2005) 119–139.

[11] M. R. Garey, D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, New York, 1979.

[12] J.-C. Picard, M. Queyranne, The time-dependent traveling salesman problem and its application to the tardiness problem in one-machine scheduling, Operations research 26 (1978) 86–110.

[13] L. Gouveia, S. Voß, A classification of formulations for the (time-dependent) traveling salesman problem, European Journal of Operational Research 83 (1995) 69–92.

[14] C. Malandraki, R. B. Dial, A restricted dynamic programming heuristic algorithm for the time dependent traveling salesman problem, European Journal of Operational Research 90 (1996) 44–55.

[15] S. Ichoua, M. Gendreau, J. Potvin, Vehicle dispatching with time-dependent travel times, European Journal of Operational Research 144 (2003) 379–396.

[16] T. Ibaraki, S. Imahori, M. Kubo, T. Masuda, T. Uno, M. Yagiura, Effective local search algorithms for routing and scheduling problems with general time-window constraints, Transportation Science 39 (2005) 206–232.

[17] S. Hart, M. Sharir, Nonlinearity of davenport-schinzel sequences and of generalized path compression schemes, Combinatorics 6 (1986) 151–177.

[18] S. Lin, Computer solutions of the traveling salesman problem, Bell System Technical Journal 44 (1965) 2245–2269.

[19] I. Or, Traveling salesman-type combinatorial problems and their relation to the logistics of regional blood banking, Ph.D. thesis, Department of Industrial Engineering and Management Sciences Northwestern University, Evanston, IL (1976).

[20] S. Reiter, G. Sherman, Discrete optimizing, J. Society for Industrial and Applied Mathematics 13 (1965) 864–889.

[21] M. M. Solomon, The vehicle routing and scheduling problems with time window constraints, Operations Research 35 (1987) 254–265.

[22] J. Homberger, H. Gehring, A two-phase hybrid metaheuristic for the vehicle routing problem with time windows, European Journal of Operational Research 162 (2005) 220–238.

[23] T. Ibaraki, S. Imahori, K. Nonobe, K. Sobue, T. Uno, M. Yagiura, An iterated local search algorithm for the vehicle routing problem with convex time penalty functions, submitted for publication.

[24] J. Homberger, H. Gehring, Two evolutionary metaheuristics for the vehicle routing problem with time windows, INFOR 37 (1999) 297–318.

[25] H. Gehring, J. Homberger, Parallelization of a two-phase metaheuristic for routing problems with time windows, Journal of Heuristics 8 (2002) 251–276.

[26] J. Berger, M. Barkaoui, O. Bräysy, A route-directed hybrid genetic approach for the vehicle routing problem with time windows, INFOR 41 (2003) 179–194.

[27] O. Bräysy, A reactive variable neighborhood search for the vehicle routing problem with time windows, INFORMS Journal on Computing 15 (2003) 347–368.

[28] R. Bent, P. V. Hentenryck, A two-stage hybrid local search for the vehicle routing problem with time windows, Transportation Science 38 (2004) 515–530.

[29] H. Gehring, J. Homberger, A parallel hybrid evolutionary metaheuristic for the vehicle routing problem with time windows, in: Proceedings of EUROGEN99, University of Jyväskylä, 1999, pp. 57–64.

[30] H. Li, A. Lim, Large scale time-constrained vehicle routing problems: a general metaheuristic framework with extensive experimental results, AI Review.

[31] A. L. Bouthillier, T. Crainic, A hybrid genetic algorithm for the vehicle routing problem with time windows, Tech. rep., Centre for Research on Transportation, University of Montreal, Canada (2003).

[32] O. Bräysy, G. Hasle, W. Dullaert, A multi-start local search algorithm for the vehicle routing problem with time windows, European Journal of Operational Research 159 (2004) 586–605.

[33] D. Mester, O. Bräysy, Active guided evolution strategies for large-scale vehicle routing problems with time windows, Computers and Operations Research 32 (2005) 1593–1614.

[34] H. Hashimoto, An iterated local search algorithm for the time-dependent vehicle routing problem with time windows, Master's thesis, Department of Applied Mathematics and Physics, Graduated School of Informatics, Kyoto University (March 2005).