Efficient Move Evaluations for Time-Dependent Vehicle Routing Problems

Thomas R. Visser*, Remy Spliet

Econometric Institute, Erasmus University Rotterdam, P.O. Box 1738, 3000 DR Rotterdam, The Netherlands

Last-Mile Logistics Research Group, Erasmus Research Institute of Management, Erasmus University Rotterdam, P.O. Box 1738, 3000 DR Rotterdam, The Netherlands

EI2017-23

Abstract

In this paper we introduce several new methods for efficiently evaluating moves in neighborhood search heuristics for routing problems with time-dependent travel times. We consider both the case that route duration is constrained and the case that route duration appears in the objective. We observe that the composition of piecewise linear functions can be evaluated in various orders when computing the route duration. We use this to develop a new tree based data structure to improve the complexity of computations and memory usage. This also allows us to present methods that have the best known computational complexity, while they do not even require a lexicographic order of search. Our numerical experiments illustrate the trade-off between computation time and memory usage among the different methods. On 1000 customer instances, our methods are able to speed-up a construction heuristic by up to 8.89 times and an exchange neighborhood improvement heuristic by up to 3.94 times, without requiring excessive amounts of memory.

Keywords: Vehicle Routing Problems, Neighborhood Search, Feasibility Check, Time-dependent Travel Times, First-in-first-out, Duration constraints

1. Introduction

The classic Vehicle Routing Problem with Time Windows (VRPTW) consists of finding a set of routes satisfying all customer requests within their time windows using a homogeneous fleet of vehicles with limited capacity while minimizing total travel costs. Recently, much attention has been given to routing problems in which travel times are assumed to be time-dependent, see for instance [1, 3, 6, 7, 14, 15, 18, 19] and see Gendreau et al. [11] for a recent literature review. Also in recent works on various orienteering problems, time-dependent travel times appear [8, 9, 23], see Gunawan et al. [13] for a recent survey. Time-dependent travel times are important in many real world applications, for instance to model road congestion or public transportation networks [11, 13]. In many studies, the total route duration, including waiting time, is minimized (see for example [1, 3, 18, 19]) or constrained (see for example [8, 9, 23]). Route duration in the objective can model a driver's salary, while the constrained route duration can model the maximum allowed working time of a driver.

Although exact methods have been proposed in the literature to solve time-dependent routing problems, for instance Dabia et al. [3] which solve some instances up to 100 requests to optimality, (meta-)heuristics are needed to obtain high quality solutions for real world instances with 1000+ requests. Many heuristics for solving various rich vehicle routing problems rely internally on some form of Neighborhood Search [24], see for example [1, 6, 7, 8, 9, 14, 23]. Typically, a family of neighboring solution schedules, generated by applying various moves on the current incumbent solution schedule, are iteratively checked for feasible improvements.

*Corresponding author.
Email addresses: t.r.visser@ese.eur.nl (Thomas R. Visser), spliet@ese.eur.nl (Remy Spliet)

In such algorithms, it is critical to quickly check feasibility and the objective value of these moves. It is known that given a route as sequence of requests, feasibility in the duration constrained VRPTW (so without time-dependent travel times) [2, 22] can be checked in $\mathcal{O}(n)$ time [25]. This is also the case for the time-dependent VRPTW (so without duration constraints and without duration objective) [6, 25]. However, when route duration and time-dependent travel times must be simultaneously optimized, the problem becomes more difficult.

The use of precalculated values stored as global data can be effective to speed-up Neighborhood Search procedures by avoiding unnecessary re-calculations during move evaluations. Kindervater and Savelsbergh [17] proposed a framework to store global variables related to time windows and capacity constraints of a route in memory. Moves are evaluated in a so-called lexicographic order such that these global variables can be updated efficiently during the search. Many authors have since published effective global route variables for many different routing and scheduling problems, including Campbell and Savelsbergh [2] who proposed global data for many constrains including shift time-limits. Recently, Vidal et al. [25] surveyed and generalized this concept for many timing subproblems. This generalization is called "Reoptimization by concatenation". Using this framework, move evaluations of the duration minimized or constrained VRPTW [2, 22] and the time-dependent VRPTW (without duration constraints) [6] can be done in $\mathcal{O}(1)$ time. However, Vidal et al. [25] note that to their knowledge no efficient method for reoptimization exists for the move evaluation of the duration constrained or minimizing time-dependent VRPTW.

Hashimoto et al. [14] discuss efficient move evaluations for the time-dependent VRPTW with additionally time-dependent piecewise linear start of service costs. We notice that the time-dependent VRPTW with route duration constraints or minimization can be reformulated to the problem of Hashimoto et al. [14], thus allowing efficient reoptimization techniques to be applied. However, detailed analysis of reoptimization methods specifically for the time-dependent VRPTW with route duration constraints or minimization reveals new insights to increase performance further, which is the main focus of this paper.

The contributions in this paper are the following. We show that the *earliest-* and *latest arrival time* global variables of Savelsbergh [22], Campbell and Savelsbergh [2] can be generalized to piecewise linear forward- and backward *ready time* functions, and we prove that move evaluations using these stored functions in reoptimization can be done in $\mathcal{O}(np)$ time, which is an improvement over a naive approach which takes $\mathcal{O}\big(n^2p\big)$ time. Here n is the total number of customers and p the maximum number of breakpoints of a travel time function. Furthermore, we prove that the feasibility and cost calculation of a route without precalculations can be done in $\mathcal{O}(np\log n)$ time, improving the previously known $\mathcal{O}\big(n^2p\big)$ time. Using these ideas, we propose a novel data structure which is small in memory, $\mathcal{O}(np\log n)$, but allows the move evaluation complexity to remain of $\mathcal{O}(np)$, even when the neighborhood is searched in non-lexicographic order. This turns out to be particularity useful for evaluating advanced neighborhoods such as k-exchange. We support our complexity results by presenting numerical results on large benchmark instances of 1000 customers. Furthermore, we illustrate the general applicability of the speed-up methods by discussing extensions such as Multiple Time Windows.

This paper is structured as follows. Section 2 contains the Time-dependent VRPTW with route duration constraints and minimization, and a typical Neighborhood Search procedure for solving it. In Section 3, we introduce the concept of ready time functions, which are the main ingredient of our speed-up methods. In Section 4, we discuss a speed-up method using forward and backward ready time functions, while in Section 5 we introduce a new data structure consisting of ready time function trees. We discuss in Section 6 some additional methods and in Section 7 we provide a summary of all methods. Section 8 contains the results of our computational experiments and in Section 9 we illustrate the general applicability of the methods by providing some applications. Section 10 contains our conclusions.

2. Time-dependent VRPTW

The Time-Dependent VRPTW (TDVRPTW), with route duration constraints and minimization, is defined on a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$, with vertex set $\mathcal{V} = \mathcal{V}^{\mathrm{C}} \cup \{o, d\}$ consisting of a set of $n = \left|\mathcal{V}^{\mathrm{C}}\right|$ customer vertices and two vertices representing the depot: source vertex o and sink vertex d. There is a fleet of identical vehicles, R in total, each with a capacity Q, starting at depot vertex o and ending at depot vertex d. Let a vertex $i \in \mathcal{V}$ be characterized by its demand q_i, service time s_i and time window $[a_i, b_i]$, with a_i (b_i) the earliest (latest) time at which service can start at the location. Without loss of generality, the demands at the depot vertices satisfy $q_o = q_d = 0$. It is assumed that a vehicle arriving at a customer before the time window opens must wait at the customer's location. Furthermore, let the planning horizon be finite and given by $[0, T]$. Let $\tau_{ij}(t)$ denote the travel time function which gives the travel time from vertex i to vertex j when departing from vertex i at time $t \in [0, T]$. The travel time functions are piecewise linear, continuous and satisfy the first-in-first-out (FIFO) property, meaning that the arrival time functions $\alpha_{ij}(t) := t + \tau_{ij}(t)$ are all strictly increasing [15]. Furthermore, each travel time function has at most p breakpoints with p a fixed integer (which corresponds to at most $\lceil p/2 \rceil$ speed zones, see [12, 15]). Travel times at any time t do not have to satisfy the triangle inequality. Let c_{ij} be the fixed distance cost for traversing arc $(i, j) \in \mathcal{A}$, for instance based on distance, and let c^{DUR} be the fixed duration cost per time unit a vehicle is away from the depot. Also, the fixed distance costs c_{ij} do not have to satisfy the triangle inequality. Let Δ denote the maximum route duration in case this is constrained. The goal of the problem is to find at most R feasible vehicle routes, i.e., *od*-paths in $\mathcal{G}$, covering all customer requests with lowest total cost. Here, the total cost consists of either the sum of all distance costs c_{ij} of the arcs used, the sum of all route duration costs of the routes, or both.

We consider the above TDVRPTW to be solved heuristically by a Neighborhood Search-based method. In this paper, we study the timing subproblem of checking feasibility and objective value change of insertion- and exchange type of moves, used extensively by such Neighborhood Search methods to solve the above problem.

2.1. Neighborhood Search

A Neighborhood Search procedure typically starts with an initial solution S and considers the neighborhood of this solution, $\mathcal{N}(S)$, which contains all solutions S' which are in some sense close to S. Typical examples of such neighborhoods include insertion, swap, 2-Opt* and k-exchange [5]. By searching $\mathcal{N}(S)$, a new solution $S^* \in \mathcal{N}(S)$ is found that is feasibile and has the lowest cost. We *move* to this new solution by replacing S with S^*. This procedure typically repeats until the local optimum solution is found which contains no improving solution in its neighborhood. Metaheuristics provide ways to continue the search beyond a local optimum, see for instance Labadie et al. [20], but still the most time consuming part of such methods is typically the evaluation of all the moves in a neighborhood. Common speed-up techniques include the use of *pre-checks* and the use of *pre-calculations*. Pre-checks are quick calculations to conclude infeasibility or inferiority of a move without having to perform the full time-consuming exact feasibilty or cost calculations. Examples include time window violation checks using bounds on travel time and cost evaluations using bounds on the exact cost. Pre-calculations try to speed up the exact move evaluation calculations by storing partial calculation results, related to the current solution S, in memory. The partial results in memory need to be updated between the Neighborhood Search iterations to reflect the new solution. This paper focuses mainly on speed-up methods of the latter kind, although we will see that their efficiency can depend on the used pre-checks.

3. Ready time functions

Let us introduce some definitions and theorems regarding so-called ready time functions [3], which are the key ingredient of our analysis.

First, let us define the *time window ready time function* $\theta_i(t)$ for each vertex $i \in \mathcal{V}$, which represents the earliest time a vehicle is ready after service when arriving at vertex i at time t.

Definition 1 (Time window ready time function). The time window ready time function $\theta_i : [0, b_i] \to \mathbb{R}$ of vertex $i \in \mathcal{V}$ is defined as

$$\theta_i(t) := \begin{cases} a_i + s_i & \text{if } t < a_i, \\ t_i + s_i & \text{if } t \in [a_i, b_i]. \end{cases} \tag{1}$$

Time window ready time functions are a special case of the general *ready time functions*, which are convenient for determining the minimum route duration of a given route of customers. Let us define these functions as follows.

Definition 2 (Ready time function). Given a route $r = (\ldots, i, \ldots, j, \ldots)$ as a path of vertices on $\mathcal{G}$, the ready time function $\delta^r_{ij}(t)$ is defined as the function that gives the (earliest) time when *service is completed* at vertex $j \in \mathcal{V}$ while *arriving* at vertex $i \in \mathcal{V}$ at time t when executing route r.

By the FIFO property, later departures yield later arrivals at all subsequent vertices in a route. Hence, service at a vertex must start as soon as possible to minimize route duration. Since no other time-dependent penalties or constraints occur in our TDVRPTW, this uniquely defines the value of $\delta^r_{ij}(t)$ by using that every activity between i and j must start as soon as possible. Therefore, the ready time function δ^r_{ij} between any two vertices i and j in route $r = (\ldots, i, \ldots, j, \ldots)$ can be computed as follows:

$$\delta^r_{ij}(t) = \left(\theta_j \circ \alpha_{j-1,j} \circ \cdots \circ \theta_{i+2} \circ \alpha_{i+1,i+2} \circ \theta_{i+1} \circ \alpha_{i,i+1} \circ \theta_i\right)(t), \tag{2}$$

using the function composition notation $(f \circ g)(t) := f(g(t))$. Given the o–d ready time function δ^r_{od} of route r, the minimum route duration Δ^*_r can be calculated as follows:

$$\Delta^*_r = \min_{t \in T} \left\{\delta^r_{od}(t) - t\right\}. \tag{3}$$

The corresponding optimal depot departure time t^{r*}_o for route r is given by:

$$t^{r*}_o \in \arg\min_{t \in T} \left\{\delta^r_{od}(t) - t\right\}. \tag{4}$$

By the FIFO property, all ready time functions are nondecreasing. It turns out that all ready time functions are also piecewise linear and continuous but generally not convex. We will proof this formally in the next section, when we analyze the complexity of computing compositions. The minimum route duration in Equation (3) is therefore attained by at least one breakpoint of $\delta^r_{od}(t)$, which can be found in polynomial time by enumerating over the breakpoints of $\delta^r_{od}(t)$.

3.1. Complexity Analysis

Let a piecewise linear, continuous and nondecreasing function $f : [0, T_f] \to \mathbb{R}$ with domain $\operatorname{dom}(f) := [0, T_f] \subseteq [0, T]$ be given. We define its *ordered set of breakpoints* $\mathcal{F}_f := \left\{(t_1, f(t_1)), (t_2, f(t_2)), \ldots, \left(t_{\phi_f}, f(t_{\phi_f})\right)\right\}$, with a number of $\phi_f := |\mathcal{F}_f|$ breakpoints. Without loss of information, let us define $\mathcal{F}_f$ to have the special property that the first breakpoint of f, $(0, f(0))$, is omitted in the set $\mathcal{F}_f$ only if f starts with a horizontal segment, i.e., if $f(0) = f(t_1)$. For example, $\mathcal{F}_{\alpha_{ij}} = \{(0, t_{ij}), (T - t_{ij}, T)\}$ is the ordered set of breakpoints of a classical non-time dependent arrival time function $\alpha_{ij}(t)$ with constant travel time $t_{ij} < T$, while $\mathcal{F}_{\theta_i} = \{(a_i, a_i + s_i), (b_i, b_i + s_i)\}$ is the ordered set of breakpoints of a time window ready time function $\theta_i(t)$ given by Equation (1). Notice that the former function starts with a non-horizontal segment and thus its set of ordered breakpoints starts with a breakpoint at $t = 0$, while the latter function starts with a horizontal segment for $t \in [0, a_i]$, and thus its set of ordered breakpoints starts with a breakpoint at $t = a_i$. Throughout this paper, any piecewise linear, continuous and nondecreasing function f is computationally directly associated with its ordered set of breakpoints $\mathcal{F}_f$.

Theorem 3. *Let $f_1(t)$ and $f_2(t)$ be two piecewise linear, continuous and nondecreasing functions. The following properties hold for the composition $f = f_2 \circ f_1$:*

1. *f is again a piecewise linear, continuous and nondecreasing function,*
2. *The number of breakpoints ϕ_f of f is at most $\phi_{f_1} + \phi_{f_2} - 2$. Moreover, f has at most 1 breakpoint if either $\phi_{f_1} = 1$ or $\phi_{f_2} = 1$, and $\phi_f = 0$ if either $\phi_{f_1} = 0$ or $\phi_{f_2} = 0$,*
3. *Calculation of f requires at most $\mathcal{O}(\phi_{f_1} + \phi_{f_2})$ operations.*

Proof. Proof of Statement 1: The composition of two continuous functions is again continuous. Let $t \in \operatorname{dom}(f_1)$ be such that $f_1(t) \in \operatorname{dom}(f_2)$ and both t is not a breakpoint of f_1 and $f_1(t)$ is not a breakpoint of f_2. Since f_1 is differentiable at t and f_2 at $f_1(t)$, let f_1' and f_2' denote the derivatives of f_1 and f_2 respectively, between their breakpoints. By elementary calculus, it holds that the derivative f', which gives the slope of the composition, is given by $f'(t) = f_1'(t) \cdot f_2'(f_1(t))$ for any t such that both $f_1(t)$ and $f_2(f_1(t))$ are between breakpoints. Because both functions are piecewise linear, the derivatives f_1' and f_2' are constant between breakpoints of respectively f_1 and f_2. Their product is therefore constant as well between breakpoints. We conclude f is a piecewise linear function as well. Also, because both derivatives are nonnegative by the nondecreasing property of f_1 and f_2, the derivative f' is also nonnegative, and by additionally using the continuity of f it follows that f is nondecreasing.

Proof of Statement 2: By the above observations, the value of the derivative f' does not change more than $\phi_{f_1} + \phi_{f_2}$ times on its domain. Therefore, f cannot have more than $\phi_{f_1} + \phi_{f_2}$ breakpoints. Notice that the resulting domain of f will be $\operatorname{dom}(f) = [0, \min\{f_1(T_{f_1}), T_{f_2}\}]$, given that $\operatorname{dom}(f_1) = [0, T_{f_1}]$ and $\operatorname{dom}(f_2) = [0, T_{f_2}]$. This effectively reduces the number of breakpoints f can have by at least one, since the breakpoint at $\max\{f_1(T_{f_1}), T_{f_2}\}$ falls outside the resulting domain. The bound on the number of breakpoints of f can be tightened further by using properties of our breakpoint representation. Let $(t_1^1, f_1(t_1^1))$ be the first breakpoint of f_1 and $(t_1^2, f_2(t_1^2))$ the first breakpoint of f_2. If $t_1^1 > 0$, then f_1 starts with a horizontal segment, i.e., zero slope and a similar condition holds for f_2. Therefore, the composition f will start a with horizontal segment on the first part of domain $[0, \max\{t_1^1, f_1^{-1}(t_1^2)\}]$, with f_1^{-1} the inverse of f_1. The breakpoint corresponding to $\min\{t_1^1, f_1^{-1}(t_1^2)\}$ falls inside this horizontal segment at the start and is removed from the resulting breakpoint representation. This still holds when either $t_1^1 = 0$ or $t_1^2 = 0$. The bound on the number of breakpoint of f is therefore $\phi_{(f_2 \circ f_1)} \leq \phi_{f_1} + \phi_{f_2} - 2$. Naturally, the following special cases hold: $\phi_f \leq 1$ if either $\phi_{f_1} = 1$ or $\phi_{f_2} = 1$, which means that at least one of the two functions corresponds with a fixed arrival time, and $\phi_f = 0$ if either $\phi_{f_1} = 0$ or $\phi_{f_2} = 0$.

Proof of Statement 3: Given that both sets of breakpoints $\mathcal{F}_{f_1}$ and $\mathcal{F}_{f_2}$ of respectively f_1 and f_2 are sorted in time, the new set of sorted breakpoints $\mathcal{F}_f$ can be constructed from begin to end using at most $\phi_{f_1} + \phi_{f_2}$ comparisons. A single comparison is used to determine which of the first remaining breakpoint of both sets is the earliest and should be incorporated in $\mathcal{F}_f$ first. Since one comparison is responsible for incorporating one breakpoint from either $\mathcal{F}_{f_1}$ and $\mathcal{F}_{f_2}$, at most $\phi_{f_1} + \phi_{f_2}$ comparisons are needed in total. Furthermore, the calculation of the values of a new breakpoint $(t, f(t))$ of f can be done in $\mathcal{O}(1)$ time since either value t or $f(t)$ is part of a breakpoint in respectively $\mathcal{F}_{f_1}$ or $\mathcal{F}_{f_2}$ and the other value can be calculated by linear interpolation in $\mathcal{O}(1)$. Since the corresponding line segment is already known by the fact that the breakpoint sets are sorted, no additional operations are needed to locate the right line segment for interpolation. Therefore, the composition requires at most $\mathcal{O}(\phi_{f_1} + \phi_{f_2})$ operations. □

The proof of Statement 3 of Theorem 3 illustrates an efficient algorithm for calculating compositions of nondecreasing piecewise linear functions. Furthermore, the following corollary follows directly from Statement 2 of Theorem 3.

Corollary 4. *The composition $f = f_2 \circ f_1$ can only have more breakpoints than either f_1 or f_2 if both $\phi_{f_1} \geq 3$ and $\phi_{f_2} \geq 3$.*

Throughout this paper, it is assumed that for all arcs $(i,j) \in \mathcal{A}$, the number of breakpoints of the arrival time function is bounded by some fixed $p \in \mathbb{N}$: i.e., $\phi_{\alpha_{ij}} \leq p$ for all $(i,j) \in \mathcal{A}$. Recall that the number of breakpoints $\phi_{\theta_i} = 2$ for any time window ready time function θ_i. By Corollary 4 and Equation (2), taking the composition of arrival time- and time window ready time functions repeatedly can only increase the number of breakpoints of the resulting composition if $p \geq 3$. This leads to the following Lemma.

Lemma 5. *The ready time function δ_{ij}^r, with $i, j \in r$ such that $i < j$ and a total of m vertices are visited, has $\mathcal{O}(mp)$ number of breakpoints, which can be calculated from scratch in at most $\mathcal{O}(m^2 p)$ operations. In the special case of $p = 2$ breakpoints, the ready time function δ_{ij}^r has at most 2 breakpoints and requires at most $\mathcal{O}(m)$ operations to calculate from scratch.*

PROOF. Repeated application of Statement 2 and 3 of Theorem 3 on the ready time function composition of Equation (2) gives the required number of breakpoints $\mathcal{O}(mp)$ and the number of operations $\mathcal{O}\big(m^2p\big)$. In the special case of $p = 2$, e.g., as in the case of classical non-time dependent travel times, all arrival- and time window ready time functions have at most two breakpoints and thus also the composition. Therefore, in this case, a ready time function is obtained by calculating at most $\mathcal{O}(m)$ function compositions and thus the total number of operations required is $\mathcal{O}(m)$. □

4. Forward and backward ready time functions

In this section, we describe a procedure that allows fast feasibility checks for insertion and exchange moves in local search procedures. The procedure is essentially a generalization of the *forward* (*backward*) *slack* variables introduced by Savelsbergh [22]. In the comprehensive survey of Vidal et al. [25], the authors state that they are unaware of efficient feasibility checks for the time-dependent VRPTW with route duration costs. However, we notice that this problem can be reformulated as a time-dependent VRPTW with linear time-dependent costs, for which Hashimoto et al. [14] provided fast feasibility checks. The method presented here is similar to the method of Hashimoto et al. [14], but applied specifically to our problem. Our presentation allows us later in Section 5 to introduce a new data structure which can decrease computation times further, in particular for more advanced moves like exchanges.

4.1. Insertion Moves

Let a route $r \in \mathcal{R}$ with m customers be given. Let us conveniently label the customers such that $r = (o, 1, 2, \dots, m, d)$. We will proof that evaluating the insertion of another customer j directly before customer i into this route, resulting in route $\tilde{r} = (o, 1, 2, \dots, i-1, j, i, \dots, m, d)$, can be done in $\mathcal{O}(mp)$ time.

First, notice that both feasibility of the vehicle capacity and the new fixed arc cost component $\sum_{(i,j)\in\tilde{r}} c_{ij}$ can be determined in $\mathcal{O}(1)$ time, given that the used capacity and fixed arc cost component of the old route r are stored in memory. The difficult part is to determine feasibility of the time window constraints and the new route $\tilde{r}$ minimum route duration $\Delta^*_{\tilde{r}}$. The latter is needed to check feasibility of the route duration constraint and to determine the new route duration cost.

A direct consequence of Equation (2) are the following equations for the o–d ready time function $\delta^r_{od}(t)$ of the old route r and the o–d ready time function $\delta^{\tilde{r}}_{od}(t)$ of the new route $\tilde{r}$.

$$\begin{aligned}
\delta^r_{od}(t) &= (\theta_d \circ \alpha_{md} \circ \cdots \circ \theta_i \circ \alpha_{i-1,i} \circ \theta_{i-1} \circ \cdots \circ \alpha_{12} \circ \theta_1 \circ \alpha_{o1} \circ \theta_o)(t) \\
&= (\delta^r_{id} \circ \alpha_{i-1,i} \circ \delta^r_{o,i-1})(t), \qquad (5)\\
\delta^{\tilde{r}}_{od}(t) &= (\theta_d \circ \alpha_{md} \circ \cdots \circ \theta_i \circ \alpha_{ji} \circ \theta_j \circ \alpha_{j,i-1} \circ \theta_{i-1} \circ \cdots \circ \alpha_{12} \circ \theta_1 \circ \alpha_{o1} \circ \theta_o)(t) \\
&= (\delta^r_{id} \circ \alpha_{ji} \circ \theta_j \circ \alpha_{j,i-1} \circ \delta^r_{o,i-1})(t). \qquad (6)
\end{aligned}$$

Equation (6) shows that in order to calculate the new route o–d ready time function $\delta^{\tilde{r}}_{od}(t)$, it suffices to calculate a composition consisting of the old route r ready time functions $\delta^r_{o,i-1}(t)$ and $\delta^r_{id}(t)$, the arrival-time functions $\alpha_{i-1,j}(t)$ and $\alpha_{ji}(t)$, and the time window ready time $\theta_j(t)$. Ready time functions of the form $\delta^r_{o,i-1}(t)$ and $\delta^r_{id}(t)$ are called respectively the *forward* and *backward* ready time functions. The equation shows that if these forward and backward ready time function are in memory, the calculation of $\delta^{\tilde{r}}_{od}(t)$ can be done by calculating four function compositions and using Equation (3) on the resulting $\delta^{\tilde{r}}_{od}(t)$ to get the exact minimum duration of the new route $\tilde{r}$.

By using Lemma 5 and Theorem 3 on Equation (6) (bottom part), one can proof the following complexity result of the exact insertion move evaluation.

Theorem 6. *Given the forward- and backward ready time functions $\delta^r_{o,i-1}(t)$ and $\delta^r_{id}(t)$ of a route $r = (o, 1, \dots, i-1, i, \dots, m, d)$, the o–d ready time function $\delta^{\tilde{r}}_{od}(t)$ of a new route $\tilde{r} = (o, 1, \dots, i-1, j, i, \dots, m, d)$ can be calculated using at most $\mathcal{O}(mp)$ operations in which also the exact minimum duration $\Delta^*_{\tilde{r}}$ of route $\tilde{r}$ is calculated. In the special case of $p = 2$, only at most $\mathcal{O}(1)$ operations are required.*

Proof. According to Lemma 5 the composition $\delta_{od}^{\tilde{r}}(t) = \left(\delta_{id}^{r} \circ \alpha_{ji} \circ \theta_j \circ \alpha_{i-1,j} \circ \delta_{o,i-1}^{r}\right)(t)$ can be calculated in $\mathcal{O}(mp)$ operations and has $\mathcal{O}(mp)$ breakpoints, if $p \geq 3$. Determining the minimum route duration $\Delta_{\tilde{r}}^{*}$ by means of Equation (3) requires additionally $\mathcal{O}(mp)$ operations, yielding a total time complexity of $\mathcal{O}(mp)$. In the special case of $p = 2$, both $\delta_{o,i-1}^{r}$ and δ_{id}^{r} have at most $p = 2$ breakpoints and thus the composition $\delta_{od}^{\tilde{r}}(t) = \left(\delta_{id}^{r} \circ \alpha_{ji} \circ \theta_j \circ \alpha_{i-1,j} \circ \delta_{o,i-1}^{r}\right)(t)$ can be calculated and used to determine minimum route duration in total of $\mathcal{O}(1)$ time. □

Notice that in the special case of $p = 2$, which includes the (classical) non-time dependent duration minimizing or constraint VRPTW, insertions can be checked in $\mathcal{O}(1)$ time using this method. This matches the well known result of Savelsbergh [22] and Campbell and Savelsbergh [2]. Moreover, the forward- and backward ready time functions, which contain at most 2 breakpoints in this special case, can be directly related to the global variables used by Savelsbergh [22] and Campbell and Savelsbergh [2] to quickly evaluate moves.

Provided that the number of customers in a route is of $\mathcal{O}(n)$, with n the total number of customers, the composition can be calculated in $\mathcal{O}(np)$ time and has $\mathcal{O}(np)$ number of breakpoints. We believe it is unlikely that for our setting a method exists which can exactly check insertion moves faster than $\mathcal{O}(np)$. Going over the breakpoints of an o–d ready time function $\delta_{od}^{\tilde{r}}$, as required to determine the minimum route duration, alone already takes $\mathcal{O}(np)$ time by breakpoint enumeration, which seems necessary for general non-convex ready time functions.

4.2. Exchange Moves

Ready time functions can also be used to quickly evaluate more advanced moves than insertion, like the commonly used exchange moves. An exchange move takes two subsequences of customers from two routes and exchanges them. Usually, the size of the subsequences considered is $0, 1, 2, \ldots, k$ with a constant maximum size $k \in \mathbb{N}$. This way, the exchange neighborhood consists of $\mathcal{O}\big(n^2k^2\big)$ possible moves. An example of an exchange move is given in Figure 1. The special case of an exchange move with $k = 1$ is

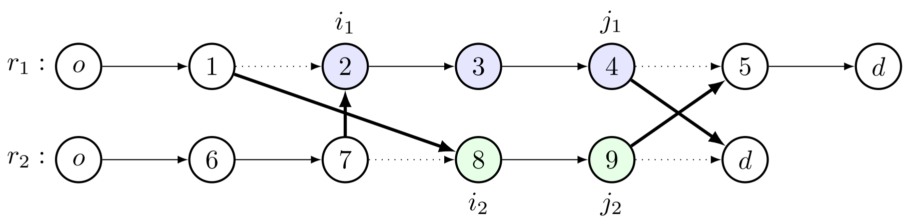


Figure 1: Illustration of the evaluation of a 3-exchange move $(i_1, j_1, i_2, j_2) = (2, 4, 8, 9)$, which exchanges customers 2, 3 and 4 from route r_1 with customers 8 and 9 from route r_2.

generally called a swap move. Also insert and 2-opt* moves can be seen as special cases of an exchange move if some subsequences are allowed to be empty.

Let us denote the move $\mathcal{M}$ which exchanges customers $i_1, \ldots, j_1$ from route r_1 with customers $i_2, \ldots, j_2$ from route r_2 by $\mathcal{M} = (i_1, j_1, i_2, j_2)$. To evaluate the move (i_1, j_1, i_2, j_2), first the o–d ready time functions $\delta_{od}^{\tilde{r}_1}$, $\delta_{od}^{\tilde{r}_2}$ resulting from the exchange need to be calculated:

$$\begin{aligned}
\delta_{od}^{\tilde{r}_1}(t) &= \left(\delta_{j_1+1,d}^{r_1} \circ \alpha_{j_2,j_1+1} \circ \delta_{i_2 j_2}^{r_2} \circ \alpha_{i_1-1,i_2} \circ \delta_{o,i_1-1}^{r_1}\right)(t), \\
\delta_{od}^{\tilde{r}_2}(t) &= \left(\delta_{j_2+1,d}^{r_2} \circ \alpha_{j_1,j_2+1} \circ \delta_{i_1 j_1}^{r_1} \circ \alpha_{i_2-1,i_1} \circ \delta_{o,i_2-1}^{r_2}\right)(t). \qquad (7)
\end{aligned}$$

Similar to Theorem 6, these compositions can be calculated and checked for minimum route durations in $\mathcal{O}((m_1 + m_2)p)$ time, with m_1 and m_2 the number of customers of routes r_1 and r_2 respectively, provided that all partial ready time functions, including the middle parts $\delta_{i_1 j_1}^{r_1}$ and $\delta_{i_2 j_2}^{r_2}$, are already available in memory. Supposing that the number of customers in the routes is of the order $\mathcal{O}(n)$, the composition can be calculated and checked for minimum route durations in $\mathcal{O}(np)$ time. However, if some functions are not in memory, then by Lemma 5 it requires $\mathcal{O}(n^2p)$ operations to calculate the missing ready time functions from scratch. This increase illustrates the benefit of having the ready time functions available in memory.

4.3. Updates and Memory

After a neighborhood is searched and the best (improving) move is found, this move is executed. The global data structures need to be updated for the changed route(s). In general, most forward (δ^r_{oi}) and backward (δ^r_{id}) ready time functions of a changed route r need to be updated, requiring $\mathcal{O}(n^2p)$ memory and time in total by Lemma 5. Would additionally all partial ready time functions (δ^r_{ij} for all $i, j \in r \backslash \{o, d\}$, $i < j$) be stored in memory, to provide instant availability of the middle segment ready time function in the exchange neighborhood, then both the update time and memory complexity increases to $\mathcal{O}\Big(\sum_{j=1}^{n} \sum_{i=j}^{n} ip\Big) = \mathcal{O}(n^3p)$. In practical settings this complexity is usually too computationally expensive. However, we can search an exchange neighborhood efficiently with only the forward and backward ready time functions in memory, requiring only $\mathcal{O}(n^2p)$ memory and update time in total, by using Lexicographic Search [17] as explained in the next section.

4.4. Lexicographic Search

Lexicographic Search entails searching a neighborhood in such an order that calculations done for evaluating a move can be used for efficient evaluation of the next move. For example, an exchange neighborhood can be searched in such an order that only relatively small computations are needed to update readily calculated middle segment ready time functions (and forward segment ready time functions) between moves. To illustrate this, let us consider two consecutive moves, $\mathcal{M}_1$ and $\mathcal{M}_2$, in an exchange neighborhood and let $\mathcal{M}_1$ be given by (i_1, j_1, i_2, j_2) in notation used earlier. Suppose we restrict the next move $\mathcal{M}_2$ to be *near* $\mathcal{M}_1$, meaning it is obtained by only extending one of the middle segments by one customer, i.e., $(i_1, j_1 + 1, i_2, j_2)$ or $(i_1, j_1, i_2, j_2 + 1)$, or by starting a new middle segment of zero or one vertex. In the first case, the middle segment ready time function $\delta^{r_1}_{i_1,j_1+1}$ can be obtained in $\mathcal{O}(np)$ time by extending the previous middle segment ready time function with one vertex: $\delta^{r_1}_{i_1,j_1+1} = \big(\theta_{j_1+1} \circ \alpha_{j_1,j_1+1} \circ \delta^{r_1}_{i_1,j_1}\big)$, which requires $\mathcal{O}(np)$ operations. In the last case of starting a new middle segment of zero or one vertex, the middle segment ready time function can also be obtained in $\mathcal{O}(np)$ operations. Together with the forward and backward ready time functions in memory, the total time required for evaluating an exchange move is still $\mathcal{O}(np)$, without needing to pre-calculate and store all partial ready time functions δ^r_{ij}. Since such an lexicographic ordering exists to cover the full exchange neighborhood, it can be searched efficiently. Notice that this does require us to keep the middle segment updated between exchange moves, which requires some computation time. Also notice that some other neighborhoods, like exchange with fixed subsequence length $k \geq 3$ of both segments, cannot be searched lexicographically. Therefore, moves in such neighborhoods generally cannot be evaluated in $\mathcal{O}(np)$ time. In Section 5 we introduce a special data structure of ready time functions which can overcome this issue.

4.5. Summary

We have seen that storing the forward- and backward ready time functions δ^r_{oi} and δ^r_{id} of all routes requires $\mathcal{O}(n^2p)$ memory and operations to update. This enables us to do fast insertion move evaluations in $\mathcal{O}(np)$ time. However, exchange moves also require the middle segment ready time functions to be available. By searching the exchange neighborhood in lexicographic order, the middle segments are updated gradually between moves which keeps the time of the move evaluation of $\mathcal{O}(np)$. Searching the exchange neighborhood in a non-lexicographic order increases the move evaluation time to $\mathcal{O}(n^2p)$, or requires us to store all partial ready time functions which cost $\mathcal{O}(n^3p)$ memory and operations to update.

5. Ready time function tree

In this section, we show that *o*–*d* ready time functions can actually be calculated from scratch in $\mathcal{O}(np \log n)$ operations, instead of the previous $\mathcal{O}(n^2p)$ operations. This insight leads to a new data structure. By storing specific partial ready time functions in a balanced binary search tree data structure, any partial ready time function can be obtained in $\mathcal{O}(np)$ operations without the need for a lexicographic order. Furthermore, we show that such trees require only $\mathcal{O}(np \log n)$ memory and operations to update, which is less than the $\mathcal{O}(n^2 p)$ memory and operations needed for the forward and backward ready time functions.

5.1. Motivation

Our motivation for an efficient tree data structure comes from a simple observation. Although the *order* in which the ready time function compositions are calculated obviously does not influence the final result, the order does significantly impact the (worst-case) number of operations required. Let us illustrate this by the following example. Suppose the *o–d* ready time function δ^r_{od} of a route $r = (o, 1, 2, 3, d)$ containing 5 vertices, $m = 3$ customers, needs to be calculated from scratch. Let us calculate it in two ways:

$$\begin{aligned}
\delta^r_{od} &= (\theta_d \overset{8}{\circ} (\alpha_{3d} \overset{7}{\circ} (\theta_3 \overset{6}{\circ} (\alpha_{23} \overset{5}{\circ} (\theta_2 \overset{4}{\circ} (\alpha_{12} \overset{3}{\circ} (\theta_1 \overset{2}{\circ} [\alpha_{o1} \overset{1}{\circ} \theta_o]))))))) && (8)\\
&= ([(\theta_d \overset{5}{\circ} \alpha_{3d}) \overset{7}{\circ} (\theta_3 \overset{4}{\circ} \alpha_{23})] \overset{8}{\circ} [(\theta_2 \overset{3}{\circ} \alpha_{12}) \overset{6}{\circ} (\theta_1 \overset{2}{\circ} \alpha_{o1} \overset{1}{\circ} \theta_o)]), && (9)
\end{aligned}$$

in which the number above the composition symbol represents the order of evaluation (composition 1 is evaluated first, composition 2 second, etc.). Equation (8) starts by calculating $\delta^r_{o1} = ((\theta_o \circ \alpha_{o1}) \circ \theta_1)$ and then extends this function by forward compositions to form δ^r_{o2}, then to δ^r_{o3}, etc., and repeats this process until δ^r_{od} is obtained. Equation (9) uses a different evaluation order. First, all functions of the form $\delta^r_{i,i+1} = (\alpha_{i,i+1} \circ \theta_{i+1})$ are calculated. Then, two neighboring functions are combined into functions of the form $\delta^r_{2i,2i+2}$, and then these latter functions are combined to functions of the form $\delta^r_{4i,4i+4}$, etc., and this is repeated until the only two remaining functions are combined to form δ^r_{od}.

Both Equations (8) and (9) require an equal number of compositions and produce the same *o–d* ready time function with at most $4p-6$ breakpoints, but for $p \geq 3$, the first equation requires much more operations in the worst case than the second. This is due to the favorable order in which the second equation evaluates the compositions. Each time subsequently the composition involving functions with the least amount of breakpoints is evaluated, while the first equation keeps evaluating the composition involving the largest function. This is reflected by the order of the number of operations required. Using Equation (8), $\mathcal{O}(ip)$ operations are required to extend $\delta^r_{o,i-1}$ to δ^r_{oi}. Therefore, in total $\mathcal{O}\left(\sum_{i=1}^{m+1} ip\right) = \mathcal{O}(m^2 p)$ number of operations are required to obtain δ^r_{od}. Using Equation 9, $\mathcal{O}(p)$ operations are required for evaluating each lowest-level compositions, compositions $1, 2, \ldots, 5$, which is in total $\mathcal{O}(mp)$. The higher level compositions 6 and 7 each require $\mathcal{O}(2p)$ operations, which is in total again $\mathcal{O}(mp)$. The highest level composition 8 requires $\mathcal{O}(4p) = \mathcal{O}((m+1)p)$ operations. In this way, the compositions are grouped in a number of $\mathcal{O}(\log m)$ levels each requiring a total of $\mathcal{O}(mp)$ operations. Overall, using Equation (9) thus requires $\mathcal{O}(mp \log m)$ operations. This is an improvement over using Equation (8) requiring $\mathcal{O}(m^2 p)$ operations.

Besides lowering the complexity of calculating an *o–d* ready time function from scratch, this favorable order of composition evaluation also gives rise to a new data structure. The intermediate ready time functions obtained during the evaluation using Equation (9), can be stored in memory. These functions form a balanced binary search tree data structure of ready time functions of a route. This data structure can be used to quickly obtain partial ready time functions.

In the following, we will formally define the ready time function tree and provide construction and memory complexity results. Furthermore, we derive complexity results for obtaining partial ready time functions and show how this is useful for checking insertion moves and the more advanced exchange moves. We conclude by elaborating its benefits in terms of non-lexicographical neighborhood searches.

5.2. Tree definition and construction

Let a route $r = (o, 1, 2, \ldots, m, d)$ be given. The ready time function tree $\mathcal{T}^r$ of route r consists of all intermediate partial ready time functions resulting from calculating δ^r_{od} in the efficient way illustrated by Equation (9). It is convenient to use the following slightly different ready time function definition δ'^r_{ij}:

$$\delta'^r_{ij} = \begin{cases} (\theta_j \circ \alpha_{j-1,j} \circ \cdots \circ \theta_1 \circ \alpha_{o,1} \circ \theta_o) & \text{if } i = o, \\ (\theta_j \circ \alpha_{j-1,j} \circ \cdots \circ \theta_{i+1} \circ \alpha_{i,i+1}) & \text{otherwise.} \end{cases} \tag{10}$$

This ensures the nice property that ${\delta'}^r_{ij} = {\delta'}^r_{lj} \circ {\delta'}^r_{il}$ for all $i < l < j$. Notice that ${\delta'}^r_{od} = \left({\delta'}^r_{md} \circ \cdots \circ {\delta'}^r_{12} \circ {\delta'}^r_{o1}\right) = \delta^r_{od}$.

Tree $\mathcal{T}^r$ is a balanced binary tree. Each leaf node of $\mathcal{T}^r$ contains a ready time function between two consecutive vertices: the leftmost leaf node contains ${\delta'}^r_{o1}$, the one next ${\delta'}^r_{12}$, etc. Leaf node i contains ${\delta'}^r_{i,i+1}$ for $i \in r \setminus \{d\}$. Each internal node of $\mathcal{T}^r$ consists of the composition of its two child nodes, for instance internal node ${\delta'}^r_{o2}$ consists of the composition of its children ${\delta'}^r_{o1}$ and ${\delta'}^r_{12}$. By construction, the root node of tree $\mathcal{T}^r$ contains the *o*–*d* ready time function $\delta^r_{od} = {\delta'}^r_{od}$. Figure 2 provides an example of the balanced binary search tree of ready time functions for a route of 15 customers.

The tree is most efficiently constructed from bottom to top, like in the example of Equation (9). First, all ready time functions ${\delta'}^r_{i,i+1}$ concerning only two adjacent route vertices are calculated and put into the leaf nodes. Next, two leaf nodes are combined by composition to form their parent node. When all parents of the leaf nodes are calculated, they are combined to form their parents. This process repeats until the last two remaining nodes are combined to form the root node, which corresponds to ${\delta'}^r_{od}$. The speed and memory performance of the construction process are summarized by the following theorem.

Theorem 7. *Without any pre-calculations, the construction of the ready time function tree $\mathcal{T}^r$ of a route r with m customers requires $\mathcal{O}(mp \log m)$ operations and equal memory to store, in case $p \geq 3$.*

PROOF. Let m be the number of customers on route r. The calculation of each leaf node ready time function ${\delta'}^r_{i,i+1}$ requires at most $p+2$ operations, except for ${\delta'}^r_{0,1}$ which requires at most $p+2+p+2$ operations. Each leaf node has a ready time function with at most p breakpoints and there are $m+1$ leaf nodes. Thus a total of at most $\mathcal{O}((m+2)p)$ operations are needed for the lowest level of the tree. By construction, the binary tree has a height of $\lceil \log_2 (m+1) \rceil + 1 = \mathcal{O}(\log m)$ levels. Let the lowest level containing only the leaf nodes be denoted by level 0 and the highest level containing the root node be level $L = \lceil \log_2 (m+1) \rceil$. The tree has at most 2^{L-l} nodes at level $l \in \{0, 1, \ldots, L\}$ and each node has a ready time function of at most $2^l p$ breakpoints. To calculate the ready time function of a node at level l requires the composition of its two childs in level $l-1$, which requires at most $2 \cdot 2^{l-1} p = 2^l p$ operations. Thus in total for level l at most $2^{L-l} 2^l p = \mathcal{O}((m+1)p) = \mathcal{O}(mp)$ operations are needed. Since there are $\lceil \log_2 (m+1) \rceil + 1$ levels, the total construction time of the tree is bounded by $(\lceil \log_2 (m+1) \rceil + 1) \cdot (m+2)p = \mathcal{O}(mp \log m)$. This is also the amount of memory needed by a very similar argument. $\square$

Notice that the *o*–*d* ready time function δ^r_{od} is always the root node in the ready time function tree and the minimum duration can be obtained from it by examining all its $\mathcal{O}(mp)$ breakpoints. Therefore, by Theorem 7, we can calculate the minimum duration of a route r with m nodes from scratch in $\mathcal{O}(mp \log m)$ operations by constructing the ready time function tree. This improves the previously best known methods requiring $\mathcal{O}(m^2 p)$ operations. Also, the memory and update time required to store and update the data structure is $\mathcal{O}(mp \log m)$, which is again lower than the memory and update time required for storing and updating all forward and backward ready time functions δ^r_{oi} and δ^r_{id}, which is $\mathcal{O}(m^2 p)$. Moreover, a tree in memory is particularly useful in obtaining any partial ready time function of a route quickly, which is the topic of the next section.

5.3. Obtaining partial ready time functions using the tree

Fast move evaluations of both insertion and exchange moves require us to calculate or obtain some partial ready time functions (forward, backward or middle segment ready time functions). If a particular ready time function is in a ready time function tree in memory, it can be obtained immediately. For partial ready time functions not in the tree, we will show that the tree nodes can be used to calculate *any* partial ready time function δ^r_{ij} in $\mathcal{O}(\bar{m}p)$ time, with $\bar{m}$ the number of vertices between i and j.

Let us first consider the following example. Suppose we have a route $r = (o, 1, 2, \ldots, m-1, m, d)$ with $m = 15$ customers and have obtained its ready time function tree, which is illustrated in Figure 2. Suppose that to evaluate an exchange move, partial ready time function ${\delta'}^r_{1,15}$ needs to be calculated. It can been seen in Figure 2 that this ready time function is not already in the tree itself. Instead of calculating it from scratch, which requires the composition of all the nodes in the blue rectangle in

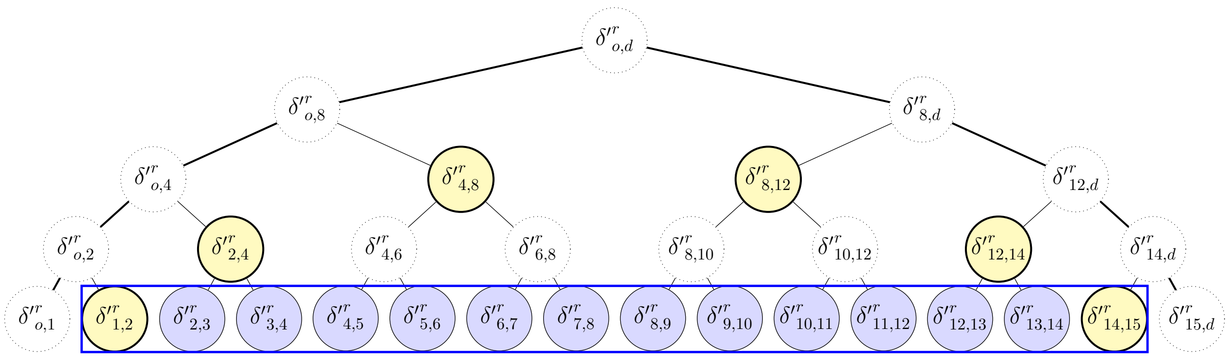


Figure 2: Example of a ready time function tree of route $(o, 1, 2, \ldots, 14, 15, d)$.

Figure 2, tree nodes at a higher level can be used to reduce the number of total operations required: $\delta'^r_{1,15} = (\delta'^r_{15,14} \circ \delta'^r_{12,14} \circ \delta'^r_{8,12} \circ \delta'^r_{4,8} \circ \delta'^r_{2,4} \circ \delta'^r_{1,2})$. In Figure 2 these corresponding tree nodes are illustrated by the yellow thick circles. The composition of these yellow nodes is most efficiently calculated using the following composition evaluation order:

$$\delta'^r_{1,15} = \left(\left[\left(\delta'^r_{14,15} \overset{2}{\circ} \delta'^r_{12,14}\right) \overset{4}{\circ} \delta'^r_{8,12}\right] \overset{5}{\circ} \left[\delta'^r_{4,8} \overset{3}{\circ} \left(\delta'^r_{2,4} \overset{1}{\circ} \delta'^r_{1,2}\right)\right]\right). \tag{11}$$

It turns out that by properties of balanced search trees, the above example is among the configurations requiring most operations. Analysis of these configurations leads to the following theorem regarding the worst-case number of operations needed for obtaining *any* partial ready time functions using the tree in memory.

Theorem 8. *Given the ready time function tree $\mathcal{T}^r$ of a route r, any partial route ready time function δ^r_{ij} can be obtained in at most $\mathcal{O}(\bar{m}p)$ operations, with $\bar{m}$ the number of vertices between i and j, inclusive, on route r.*

Proof. Suppose δ^r_{ij} needs to be obtained using the tree $\mathcal{T}^r$, with $\bar{m}$ vertices between i and j in route r. By general properties of balanced search trees (see de Berg et al. [4, p.96–99]), the ready time functions stored in the nodes of $\mathcal{T}^r$ which are most efficient for composing δ^r_{ij} (the thick yellow nodes in Figure 1) can be found as follows. Leaf node $\delta'^r_{i-1,i}$ in $\mathcal{T}^r$ corresponds to the node directly left of the required interval and leaf node $\delta'^r_{j,j+1}$ directly right next to the interval. Both these leaf nodes have a unique search path to the root node δ'^r_{od}. At some node, which we denote by δ'^r_{split}, both search paths will be merged. In Figure 1, the leaf nodes are δ'^r_{o1} and $\delta'^r_{15,d}$ and their search path merges at $\delta'^r_{\text{split}} = \delta'^r_{od}$. The most efficient nodes in the tree for the composition can now be found to be all *right* child nodes of the nodes along the (left) search path of $\delta'^r_{i-1,i}$ and all the *left* child nodes of the nodes along the (right) search path of $\delta'^r_{j,j+1}$. Here, we denote the two children of a non-leaf node in the tree as being left or right, with the left child having the ready time function with lower indices. Let us denote the composition of the right children along the left search path by δ'^{L} and likewise the composition of the left children among the right search path by δ'^{R}. Now the required ready time function can be found by calculating the composition of these two parts: $\delta'^r_{ij} = (\delta'^{\text{R}} \circ \delta'^{\text{L}})$. It can be proven (see de Berg et al. [4, p.96–99]) that for each level in the tree, at most two nodes with the same level will be part of the required composition. In case two nodes of the same level are present in the composition, one node will be contained in δ'^{L} and the other must be in δ'^{R}. Furthermore, nodes in the composition δ'^{L} increase in level with larger indices while the nodes in the composition δ'^{R} decrease in level with larger indices. Therefore, given the interval $[i, j]$ of the required ready time function and its corresponding split node δ'^r_{split}, the composition consists in worst-case of one node of *each* level below δ'^r_{split} in δ'^{L} and also one node of *each* level below δ'^r_{split} in δ'^{R}. There are $\bar{m}$ vertices between i and j, inclusive. Let $\bar{l}$ be highest level in the tree of which nodes can appear in the composition, i.e., one level below the split node. It can be seen that $\bar{l} \leq \lceil \log(\bar{m}+2) \rceil - 1$. We have seen that the order of evaluating compositions is most efficient when iteratively selecting the composition involving the smallest functions. Therefore, the compositions in ${\delta'}^{\mathrm{L}}$ are evaluated from the lowest level nodes up to level $\bar{l}$, in Figure 1 from left to right, and likewise the compositions in ${\delta'}^{\mathrm{R}}$ are evaluated, in Figure 1 from right to left, and finally ${\delta'}^{r}_{ij} = ({\delta'}^{\mathrm{R}} \circ {\delta'}^{\mathrm{L}})$ is evaluated. The total number of operations required to calculate ${\delta'}^{r}_{ij}$ in the worst case using this procedure is given by:

$$\begin{aligned}
& \mathcal{O}\left(2\sum_{l=1}^{\bar{l}-1}\sum_{k=0}^{l}2^k p + 2\sum_{k=0}^{\bar{l}-1}2^k p\right) && (12)\\
= \; & \mathcal{O}\left(2\sum_{l=2}^{\bar{l}}\left[(2^l-1)p\right] + 2(2^{\bar{l}}-1)p\right)\\
= \; & \mathcal{O}\Big(2\left(2^{\bar{l}+1}-4-\bar{l}-1\right)p + 2\left(2^{\bar{l}}-1\right)p\Big)\\
= \; & \mathcal{O}\Big(3\cdot 2^{\bar{l}+1}p - \left(\bar{l}+12\right)p\Big)\\
= \; & \mathcal{O}(3\bar{m}p)\\
= \; & \mathcal{O}(\bar{m}p).
\end{aligned}$$

In Equation (12), the double summation in the first term represents the worst-case number of operations needed to construct ${\delta'}^{\mathrm{L}}$ and is counted twice for also constructing ${\delta'}^{\mathrm{R}}$. The last term in Equation (12) corresponds to the worst-case number of operations needed to evaluate the composition ${\delta'}^{r}_{ij} = ({\delta'}^{\mathrm{R}} \circ {\delta'}^{\mathrm{L}})$. Since ${\delta'}^{r}_{ij}$ contains at most $\mathcal{O}(\bar{m}p)$ breakpoints, δ^r_{ij} can be obtained from ${\delta'}^{r}_{ij}$ in at most $\mathcal{O}(\bar{m}p)$ operations. Therefore, using the tree $\mathcal{T}^r$ in memory, δ^r_{ij} can be obtained in at most $\mathcal{O}(\bar{m}p)$ operations. $\square$

In the special case of $p = 2$, which includes the case of classical non-time dependent travel times, we have seen that all forward, backward and partial ready time functions contain at most $p = 2$ breakpoints. Therefore, the ready time tree $\mathcal{T}^r$ of a route r with m customers consists of $\mathcal{O}(m)$ tree nodes (ready time functions) of at most two breakpoints. Thus, the tree can be stored using $\mathcal{O}(m)$ memory. Also, it can be shown that construction can be done in $\mathcal{O}(m)$ time and calculation of a partial ready time function visiting $\bar{m}$ customers using the tree can be done in $\mathcal{O}(\log \bar{m})$ time. This is lower than the $\mathcal{O}(\bar{m})$ operations required when calculating such partial ready time function for $p = 2$ from scratch.

5.4. Insertion Moves

Evaluating an insertion move of inserting customer j between $i-1$ and i in route r with m customers requires calculation of $\delta^r_{o,i-1}$ and δ^r_{id}. By Theorem 8, both ${\delta'}^{r}_{o,i-1}$ and ${\delta'}^{r}_{id}$ can be calculated in $\mathcal{O}(mp)$ operations using the ready time function tree $\mathcal{T}^r$ of r. Then $\delta^r_{o,i-1} = {\delta'}^{r}_{o,i-1}$ and $\delta^r_{id} = \left(\theta_i \circ {\delta'}^{r}_{id}\right)$, with the latter requiring $\mathcal{O}(mp)$ operations. Now, the composition of the new *o*–*d* ready time function of Equation (6) is used, which again requires $\mathcal{O}(mp)$ time. In total, this method requires $\mathcal{O}(mp)$ time to evaluate an insertion move and therefore does not increase the overall complexity of the move evaluation compared to the method of Section 4.

5.5. Exchange Moves

Evaluating an exchange move which exchanges customers $i_1, \ldots, j_1$ of route r_1 with customers $i_2, \ldots, j_2$ of route r_2 requires the calculation of the following six ready time functions: $\delta^{r_1}_{o,i_1-1}$, $\delta^{r_1}_{i_1,j_1}$, $\delta^{r_1}_{j_1+1,d}$, $\delta^{r_2}_{o,i_2-1}$, $\delta^{r_2}_{i_2,j_2}$ and $\delta^{r_2}_{j_2+1,d}$. Suppose both routes have $\mathcal{O}(m)$ number of customers. By a similar argument to the insertion moves, all these ready time functions can be obtained in $\mathcal{O}(mp)$ time using the ready time function trees $\mathcal{T}^{r_1}$ and $\mathcal{T}^{r_2}$. Thus, the total complexity of evaluating an exchange move remains $\mathcal{O}(mp)$. However, this complexity does not rely on the storage of calculated middle segments for previous moves. Therefore, using the ready time function trees retains the total move evaluation complexity $\mathcal{O}(mp)$ even if the exchange neighborhood is searched in non-lexicographic order.

5.6. Updates and Memory

By Theorem 7, the tree $\mathcal{T}^r$ of a route r with m customers requires $\mathcal{O}(mp \log m)$ memory to store and an equal amount of operations to construct from scratch. Each time a move is executed which changes route r, we update its corresponding tree $\mathcal{T}^r$ by full re-construction in $\mathcal{O}(mp \log m)$ operations. This is needed, because the efficiency of obtaining partial ready time functions from the tree relies heavily on the property of the tree being balanced, i.e., having maximum height of at most $\mathcal{O}(\log m)$. It is difficult to maintain this property after the number of customers of a route has changed, without re-building the tree from scratch. Although there exist so-called *self-balancing* binary tree data structures, which can re-balance themselves after an update, we are yet to discover such a structure that truly lowers the update complexity of $\mathcal{O}(mp \log m)$ in case a move changes the number of customers in a route.

5.7. Non-lexicographic and Lexicographic Neighborhood Search

We have seen that in case of exchange moves the ready time function trees can be used to efficiently evaluate moves without requiring a particular evaluation order. This opens up possibilities for efficient non-lexicographic neighborhood searches for our problem, such as (parts of) the *Sequential Search* framework of Irnich [16] for example or a simple Variable Neighborhood Search in which the exchange subsequence length k is increased dynamically during the search.

Moreover, also in the case of Lexicographic Search the use of ready time function trees potentially decreases the average computation times. Let us illustrate this in the setting of lexicographic k-exchange. Although we have seen in Section 4.4 that the particular evaluation order of moves allows us to update middle segment ready time functions between moves with minimal effort, in practice such updates between moves are usually done in a *lazy* fashion, meaning only when this is actually needed for evaluating the new move. Quick pre-checks typically conclude infeasibility or inferiority of the new move without the need of these ready time functions. Suppose multiple consecutive moves are deemed infeasible or inferior by pre-checks. Then no time consuming exact feasibility or cost calculations are necessary and no updates of ready time functions between moves are done. When subsequently a new move passes all pre-checks, the ready time functions have to be updated, which, using regular forward extension, require a quadratic number of operations in the number of previous infeasible moves. However, when using the ready time tree these updates require only a linear number of operations in the number of previous infeasible moves. Hence, the tree method utilizes the pre-checks much better.

6. Additional methods

In this section we present two other pre-calculation methods related to the Forward and Backward (F/B) method of Section 4 and the method of Ready time function Tree (TREE) presented in Section 5: the Ready time function Tree + Forward/Backward Hybrid (TREE+F/B) and the All in memory method (ALL).

6.1. Ready time function Tree + Forward/Backward Hybrid

The combination of the forward and backward ready time functions in memory with the ready time function trees in memory is particularly useful for searching advanced Neighborhoods such as k-exchange. The needed forward and backward ready time functions are pre-calculated in memory, while the ready time function trees can be used to update middle segment ready time function efficiently. The move evaluation complexity remains $\mathcal{O}(mp)$, with m the number of customers in the affected routes. Similar to using only the ready time trees, this move complexity is maintained even if the neighborhood is searched in non-lexicographic order. The memory requirement and update complexity between iterations is $\mathcal{O}(m^2 p)$ per route, due to the complexity of the Forward and Backward method.

6.2. *All ready time functions in memory*

Another method to ensure even quicker move evaluations for advanced Neighborhoods such as k-exchange is to keep *every* partial ready time function δ^r_{ij} in memory. For each move, forward/backward and middle segment ready time functions are all pre-calculated, so only some composition of these ready time functions needs to be calculated and minimized. This still requires $\mathcal{O}(mp)$ operations with m number of customers in the affected routes, but in practice this will decrease overall neighborhood running times even further compared to the above TREE + F/B hybrid method. The price is however an increase in memory and update operations needed to maintain all these partial ready time functions: $\mathcal{O}(m^3p)$ memory and update operations are now needed per route, which is a factor m higher than the Forward and Backward method.

7. Summary of the methods

We give a brief summary of the methods considered in this paper by providing their computation time and memory complexities of insertion and k-exchange neighborhood search. Table 1 contains complexities of the following methods, with n the total number of customers and p the highest number of breakpoints among the arrival time functions.

- **non-TD**
  For comparison, we include the known complexity results of the efficient forward (backward) slack methods [2, 22] for the classical non-time dependent, duration constrained or minimized VRPTW.
- **NAIVE**
  This methods re-calculates the complete ready time function $\delta^{\tilde{r}}_{od}$ from scratch for each new move by iterative forward composition. Besides from the arrival time functions, no (other) ready time functions are stored in memory. To provide a fair comparison, this method is allowed to keep only the last calculated forward-, middle and backward segment ready time functions of the previous move in memory.
- **TREE**
  Fast insertion checks by storing a ready time function tree for each route. Forward, backward and partial ready time functions are calculated using the ready time function trees. Exchange moves can also be evaluated efficiently by using non-Lexicographic Search.
- **F/B**
  Faster insertion checks by storing forward and backward ready time functions. Exchange moves can only be evaluated efficiently by using Lexicographic Search, since the middle segment ready time functions are not in memory.
- **TREE+F/B**
  Even faster insertion checks by both storing a ready time function tree and forward and backward ready time functions for each route and calculating the needed partial ready time functions. Exchange moves can also be evaluated efficiently using non-Lexicographic Search.
- **ALL**
  Fastest insertion checks by storing all partial ready time functions. Exchange moves can also be efficiently evaluated in case of non-Lexicographic Search, but requires a higher memory and update time complexity.

The investigated methods, NAIVE, TREE, F/B, TREE+F/B and ALL, are presented in this order to illustrate their increasing amount of pre-calculation done (e.g., NAIVE doing no pre-calculations while ALL does the most), so increasing in expected move evaluation efficiency as well as needed update time and memory.

The table gives the complexities of the methods in case of *Insertion*, *k-Exchange* (Lexicographic) and *k-Exchange non-Lexicographic* Neighborhood Searches, for which we report the complexity of evaluating

Table 1: Complexities of the investigated methods during insert and exchange neighborhood evaluations.

Neighborhood	Operation	non-TD	NAIVE	TREE	F/B	TREE+F/B	ALL
Insert	single move	$\mathcal{O}(1)$	$\mathcal{O}(n^2p)$	$\boldsymbol{\mathcal{O}(np)}$	$\boldsymbol{\mathcal{O}(np)}$	$\boldsymbol{\mathcal{O}(np)}$	$\boldsymbol{\mathcal{O}(np)}$
	total	$\mathcal{O}(n^2)$	$\mathcal{O}(n^4p)$	$\boldsymbol{\mathcal{O}(n^3p)}$	$\boldsymbol{\mathcal{O}(n^3p)}$	$\boldsymbol{\mathcal{O}(n^3p)}$	$\boldsymbol{\mathcal{O}(n^3p)}$
	update/mem.	$\mathcal{O}(n)$	–	$\boldsymbol{\mathcal{O}(np\log n)}$	$\mathcal{O}(n^2p)$	$\mathcal{O}(n^2p)$	$\mathcal{O}(n^2p)$
Full constr.	total	$\mathcal{O}(n^3)$	$\mathcal{O}(n^5p)$	$\boldsymbol{\mathcal{O}(n^4p)}$	$\boldsymbol{\mathcal{O}(n^4p)}$	$\boldsymbol{\mathcal{O}(n^4p)}$	$\boldsymbol{\mathcal{O}(n^4p)}$
k-Exchange	single move	$\mathcal{O}(1)$	$\mathcal{O}(n^2p)$	$\boldsymbol{\mathcal{O}(np)}$	$\boldsymbol{\mathcal{O}(np)}$	$\boldsymbol{\mathcal{O}(np)}$	$\boldsymbol{\mathcal{O}(np)}$
	total	$\mathcal{O}(n^2k^2)$	$\mathcal{O}(n^4k^2p)$	$\boldsymbol{\mathcal{O}(n^3k^2p)}$	$\boldsymbol{\mathcal{O}(n^3k^2p)}$	$\boldsymbol{\mathcal{O}(n^3k^2p)}$	$\boldsymbol{\mathcal{O}(n^3k^2p)}$
	update/mem.	$\mathcal{O}(n)$	–	$\boldsymbol{\mathcal{O}(np\log n)}$	$\mathcal{O}(n^2p)$	$\mathcal{O}(n^2p)$	$\mathcal{O}(n^3p)$
k-Exchange	single move	$\mathcal{O}(n)$	$\mathcal{O}(n^2p)$	$\boldsymbol{\mathcal{O}(np)}$	$\mathcal{O}(n^2p)$	$\boldsymbol{\mathcal{O}(np)}$	$\boldsymbol{\mathcal{O}(np)}$
Non-lex	total	$\mathcal{O}(n^3k^2)$	$\mathcal{O}(n^4k^2p)$	$\boldsymbol{\mathcal{O}(n^3k^2p)}$	$\mathcal{O}(n^4k^2p)$	$\boldsymbol{\mathcal{O}(n^3k^2p)}$	$\boldsymbol{\mathcal{O}(n^3k^2p)}$
	update/mem.	$\mathcal{O}(n)$	–	$\boldsymbol{\mathcal{O}(np\log n)}$	$\mathcal{O}(n^2p)$	$\mathcal{O}(n^2p)$	$\mathcal{O}(n^3p)$

Table 2: Complexities of the investigated methods during insert and exchange neighborhood evaluations, when the maximum number of customers in a route is bounded by M and static move evaluations are used.

Neighborhood	Operation	non-TD	NAIVE	TREE	F/B	TREE+F/B	ALL
Insert	single move	$\mathcal{O}(1)$	$\mathcal{O}(M^2p)$	$\boldsymbol{\mathcal{O}(Mp)}$	$\boldsymbol{\mathcal{O}(Mp)}$	$\boldsymbol{\mathcal{O}(Mp)}$	$\boldsymbol{\mathcal{O}(Mp)}$
	total	$\mathcal{O}(Mn)$	$\mathcal{O}(M^3np)$	$\boldsymbol{\mathcal{O}(M^2np)}$	$\boldsymbol{\mathcal{O}(M^2np)}$	$\boldsymbol{\mathcal{O}(M^2np)}$	$\boldsymbol{\mathcal{O}(M^2np)}$
	update	$\mathcal{O}(M)$	–	$\boldsymbol{\mathcal{O}(Mp\log M)}$	$\mathcal{O}(M^2p)$	$\mathcal{O}(M^2p)$	$\mathcal{O}(M^2p)$
	memory	$\mathcal{O}(n)$	–	$\boldsymbol{\mathcal{O}(np\log M)}$	$\mathcal{O}(Mnp)$	$\mathcal{O}(Mnp)$	$\mathcal{O}(Mnp)$
Full constr.	total	$\mathcal{O}(Mn^2)$	$\mathcal{O}(M^3n^2p)$	$\boldsymbol{\mathcal{O}(M^2n^2p)}$	$\boldsymbol{\mathcal{O}(M^2n^2p)}$	$\boldsymbol{\mathcal{O}(M^2n^2p)}$	$\boldsymbol{\mathcal{O}(M^2n^2p)}$
k-Exchange	single move	$\mathcal{O}(1)$	$\mathcal{O}(M^2p)$	$\boldsymbol{\mathcal{O}(Mp)}$	$\boldsymbol{\mathcal{O}(Mp)}$	$\boldsymbol{\mathcal{O}(Mp)}$	$\boldsymbol{\mathcal{O}(Mp)}$
	total	$\mathcal{O}(Mnk^2)$	$\mathcal{O}(M^3nk^2p)$	$\boldsymbol{\mathcal{O}(M^2nk^2p)}$	$\boldsymbol{\mathcal{O}(M^2nk^2p)}$	$\boldsymbol{\mathcal{O}(M^2nk^2p)}$	$\boldsymbol{\mathcal{O}(M^2nk^2p)}$
	update	$\mathcal{O}(M)$	–	$\boldsymbol{\mathcal{O}(Mp\log M)}$	$\mathcal{O}(M^2p)$	$\mathcal{O}(M^2p)$	$\mathcal{O}(M^3p)$
	memory	$\mathcal{O}(n)$	–	$\boldsymbol{\mathcal{O}(np\log M)}$	$\mathcal{O}(Mnp)$	$\mathcal{O}(Mnp)$	$\mathcal{O}(M^2np)$
k-Exchange	single move	$\mathcal{O}(M)$	$\mathcal{O}(M^2p)$	$\boldsymbol{\mathcal{O}(Mp)}$	$\mathcal{O}(M^2p)$	$\boldsymbol{\mathcal{O}(Mp)}$	$\boldsymbol{\mathcal{O}(Mp)}$
Non-lex	total	$\mathcal{O}(M^2nk^2)$	$\mathcal{O}(M^3nk^2p)$	$\boldsymbol{\mathcal{O}(M^2nk^2p)}$	$\mathcal{O}(M^3nk^2p)$	$\boldsymbol{\mathcal{O}(M^2nk^2p)}$	$\boldsymbol{\mathcal{O}(M^2nk^2p)}$
	update	$\mathcal{O}(M)$	–	$\boldsymbol{\mathcal{O}(Mp\log M)}$	$\mathcal{O}(M^2p)$	$\mathcal{O}(M^2p)$	$\mathcal{O}(M^3p)$
	memory	$\mathcal{O}(n)$	–	$\boldsymbol{\mathcal{O}(np\log M)}$	$\mathcal{O}(Mnp)$	$\mathcal{O}(Mnp)$	$\mathcal{O}(M^2np)$

a single move, the total time for searching the entire neighborhood and the update time and memory complexity needed to update and store the data structures used by the methods between neighborhood searches. Furthermore, we included the total time needed by a cheapest insertion construction heuristic (*Full constr.*), which uses the Insertion Neighborhood iteratively to route all customers from scratch (using at most n iterations of insertion). In the table, we further assume that the number of customers m of any route is $\mathcal{O}(n)$. Complexities in bold indicate the lowest complexity among the investigated methods, where we disregard non-TD as it is not suited for the TDVRPTW.

7.1. Bounded customers per route and static move evaluations

An important feature of most large real-world routing problems is the fact the maximum number of customers in a route is implicitly bounded such that it does not scale with the total number of customers n, e.g., due to capacity constraints. Often this occurs due to limited vehicle capacity, but other constraints can also play a role. Table 2 shows the complexity results similar to Table 1, so with n the number of total customers and p the highest number of breakpoints among the arrival time functions, but now using the assumption that the maximum number of customers in a route is bounded by a constant M. The table also assumes the use of so-called *static move descriptors* [26], which, in short, means that in some neighborhood search iteration except the first, only new moves concerning at least one route which was just changed need to be actively checked, provided the best moves concerning the other routes are kept in memory. Also, when executing a move, only the data structures of the affected routes need to be updated. This strategy is particularly efficient when the maximum number of customers in a route M is small compared to the total number of customers n. We use this strategy for all our computational experiments.

8. Computational Experiments

In this section, we present the results of numerical experiments in which we apply the presented methods on several benchmark instances. The methods are tested in a construction heuristic and a neighborhood search improvement heuristic, which are both implemented in C++. All runs are executed as a single thread on an Intel® Xeon® E5-2650 v2 with 2.6 GHz (Turbo Boost up to 3.6 GHz) and 32 GB of RAM. All CPU times were measured using `std::chrono::high_resolution_clock`.

We test the speed-up methods in a parallel cheapest insertion construction heuristic and a k-exchange neighborhood search improvement heuristic. Both heuristics can be summarized as follows. Each iteration, the best feasible move is found and executed. In case of the construction heuristic, a move consists of inserting an unplanned customer into a route, while in case of the improvement method a move is a k-exchange. After the first iteration, the best move for each route–customer combination (route–route combination for the improvement heuristic) is kept in memory and only moves involving changed routes are re-calculated (static move descriptors). During move evaluation, we use the following pre-checks in this order: (1) capacity; (2) non-TD time window feasibilty; (3) cost lower bound. The capacity pre-check simply checks whether the vehicle capacity of the new route is violated. The non-TD time window feasibilty uses the non-time dependent earliest and latest arrival times based on the highest speed of each arc to check time window feasibilty. The cost lower bound uses either exact new route distances, in case of distance only based objective, or estimates the new route duration by using only highest speed along each arc and service times, in case of duration only based objective, to check inferiority of the current move compared to the best move seen so far. The exact feasibility and cost of a move are only calculated when all pre-checks are passed.

8.1. Instances

We use the Gehring and Homberger [10] instances with 1000 customers for the VRPTW for our experiments. These instances are currently the largest commonly used VRPTW instances. Each of these 60 instances are split into 6 groups of 10 instances: C1, RC1, R1, C2, RC2, R2. The first letter denotes the geographic spread of the customers, with C: clustered, RC: random-clustered, R: random. The number represents the instance type, with 1: short routes and tight time windows, 2: long routes and wide time windows.

Time-dependent travel times are added to these instances by means of the speed-profiles introduced by Figliozzi [7]. For each instance, the planning horizon $[0, T] = [0, b_d]$, with b_d the depot time window end time, is partitioned into a number of speed zones each with equal duration. Each speed zone has a constant speed factor which modifies the classical (nominal) travel times based on Euclidean distance. Table 3 shows the speed-profiles used in our computational experiments. For each speed-profile TDx, it shows the maximum number of breakpoints p, and for each of the zone end times $t \cdot b_d$ the speed factor. Speed-profile TD0 corresponds to the classical non-time dependent travel times and result in travel time functions with only $p = 2$ breakpoints. Speed-profiles TD1, TD2 and TD3 each result in travel time functions with at most $p = 10$ breakpoints and decrease the travel times on average over the whole planning horizon by 25%, 50% and 75% respectively. As in Figliozzi [7], we use the same speed profile for all arcs. Zone start and end times were rounded to the nearest integer, while arrival time functions were calculated using 5 decimal precision to avoid numerical instabilities.

Table 3: Speed Profiles

		Zones $\cdot b_d$				
	p	$[0, 0.2]$	$[0.2, 0.4]$	$[0.4, 0.6]$	$[0.6, 0.8]$	$[0.8, 1]$
TD0	2	1.00	1.00	1.00	1.00	1.00
TD1	10	1.00	1.60	1.05	1.60	1.00
TD2	10	1.00	2.00	1.50	2.00	1.00
TD3	10	1.00	2.50	1.75	2.50	1.00

8.2. Insertion Experiments

The speed-up of using TREE and F/B over NAIVE during insertion move evaluations is tested using a parallel cheapest insertion construction heuristic which builds routes from scratch. Note that insertion moves are simple and only require forward and backward ready time functions. Therefore, methods TREE+F/B and ALL will not reduce computation times further and thus are not tested here.

Table 4 shows the results of the cheapest insertion construction heuristics on the Gehring and Homberger 1000 customer instances when using insertion costs based on distance only. The table shows the instance groups with the used speed profile. The column #routes shows the number of routes of the constructed solutions, respectively, averaged over the ten instances in a group. The columns #mov., #PCfeasmov. and #feasmov. show the total number of moves evaluated, number of moves that passed all pre-checks and number of moves that were found feasible by the exact move evaluation, respectively, again averaged over the instances. The Average CPU columns show the total CPU time needed for the construction heuristic in seconds, averaged over the ten instances. The column Speed-up over NAIVE shows the CPU time speed-up factor of using TREE and F/B over NAIVE, respectively, with speed-up factor 1.00 being equal in speed.

We see that the TREE and F/B methods speed-up the construction heuristic in every instance group and every speed-profile including the non-time dependent profile TD0. Most speed-up is gained on the time-dependent instances with long routes and large time windows (C2, RC2 and R2 instances), although the differences in speed-up between the speed-profiles TD1, TD2 and TD3 are minimal.

Table 5 shows the results of the same construction heuristic experiment but with route duration as objective. Again most speed-up is gained on time-dependent instances with long routes and large time windows, but the amount of speed-up is much larger compared to the distance insertion costs used in Table 4. Using distance costs, average speed-ups of up to 5.09 are observed, while using duration costs average speed-ups of up to 8.89 are observed. Note that the cost LB pre-check is weaker in case of duration costs than for distance costs. Therefore, the fraction of moves passing the pre-checks in case of duration costs is higher and the heuristics must spend more time on the exact feasibilty and cost calculations which both the TREE and the F/B method speed up. In both tables, the F/B method outperforms the TREE method by up to 2 times and on average by 1.4. The TREE method needs to evaluate some compositions to construct the forward- and backward ready time functions, while in the F/B method these are readily available in memory. The increases in update times for both methods are much lower than the overall decreases in time needed for each iteration.

8.3. Lexicographic Exchange Experiments

The next experiments compare the average computation times of NAIVE, TREE, F/B, TREE+F/B and ALL methods in a k-exchange improvement heuristic. The heuristic is run on all Gehring and Homberger 1000 customer instances. A value of maximum subsequence length $k = 8$ is selected and the heuristic is run iteratively until the local optimum is reached. The heuristic starts with the solution obtained by the construction heuristic with distance costs as described in Section 8.2. We choose to start with these solutions over the ones constructed with duration costs, because the latter tend to have a lot more and smaller routes which limits possibilities of exchanging large subsequences of customers.

Table 6 shows the results of these runs for the different speed-profiles. The column #It presents the number of iterations of the exchange heuristic used to obtain the local minimum, averaged over the group of ten instances. The columns #mov., #PCfeasmov. and #feasmov. again show the total number of moves evaluated, number of moves that passed all pre-checks and number of moves that were found feasible by the exact move evaluation, respectively, again averaged over the instances. The columns Average CPU show the CPU time in seconds needed to obtain the local optimum and the columns Speed-up over NAIVE give the factor CPU time was reduced compared to NAIVE, each averaged over the instance group. As with the construction experiments, most speed-up of F/B, TREE and ALL methods over NAIVE is gained on instances C2, RC2 and R2 with large number of customers in the routes. Although speed-ups of up to 2.54 times occur for the classical non-time dependent profile TD0, the methods are particularly able to speed-up the time-dependent speed-profiles, showing speeds-ups up to 3.61 with the F/B method, up to 3.94 with the TREE+F/B method and up to 5.48 times with the ALL method, although differences between TD1, TD2 and TD3 are minimal.

Table 4: Construction heuristic results – distance costs – $n = 1000$ customers.

						Average CPU (s)			Speed-up o. NAIVE	
		#routes	#mov.	#PCfeasmov.	#feasmov.	NAIVE	TREE	F/B	TREE	F/B
TD0	C1	100.5	3,426,065.3	205,084.4	205,084.4	1.79	1.73	1.63	1.04	1.10
	RC1	95.5	3,588,281.3	246,653.6	246,653.6	1.84	1.75	1.63	1.06	1.13
	R1	95.4	3,595,665.4	322,749.4	322,749.4	1.94	1.82	1.67	1.07	1.16
	C2	33.9	8,956,573.0	371,908.8	371,908.8	2.76	2.08	1.80	1.33	1.54
	RC2	24.9	14,211,415.5	485,851.7	485,851.7	3.86	2.32	1.94	1.67	1.99
	R2	21.8	14,292,815.3	609,510.8	609,510.8	4.38	2.47	2.01	1.77	2.17
TD1	C1	99.2	3,451,070.5	276,156.8	241,109.7	2.40	2.14	1.83	1.12	1.31
	RC1	94.6	3,594,157.5	334,627.1	271,475.3	2.46	2.13	1.82	1.16	1.35
	R1	95.2	3,603,186.3	408,761.7	355,976.2	2.84	2.39	1.94	1.19	1.46
	C2	33.2	9,129,195.8	468,515.3	398,546.2	4.18	2.70	2.05	1.55	2.04
	RC2	24.7	14,300,147.0	690,603.4	510,306.1	7.81	3.58	2.41	2.18	3.24
	R2	21.6	14,264,599.8	843,368.2	668,166.0	9.24	4.09	2.56	2.26	3.61
TD2	C1	99.0	3,471,849.1	303,541.0	270,376.0	2.52	2.21	1.88	1.14	1.34
	RC1	94.7	3,595,981.6	370,811.0	283,539.2	2.53	2.18	1.84	1.16	1.37
	R1	95.0	3,601,883.0	445,726.9	380,396.7	2.93	2.47	1.98	1.19	1.48
	C2	32.4	9,213,043.1	523,403.2	428,760.9	4.73	2.90	2.13	1.63	2.22
	RC2	24.8	14,346,930.9	831,875.0	524,698.9	9.47	4.41	2.74	2.15	3.46
	R2	21.3	14,348,174.2	999,680.8	711,870.6	13.95	5.02	3.05	2.78	4.57
TD3	C1	98.3	3,486,147.5	337,896.2	293,191.5	2.58	2.27	1.90	1.14	1.35
	RC1	94.7	3,591,158.7	415,358.2	298,731.7	2.63	2.26	1.88	1.17	1.40
	R1	94.5	3,612,366.1	481,004.6	398,440.5	3.04	2.54	2.01	1.20	1.51
	C2	32.1	9,286,906.1	594,376.6	454,312.8	4.75	3.00	2.18	1.58	2.18
	RC2	24.7	14,314,163.0	1,015,197.1	537,624.1	17.34	5.90	3.41	2.94	5.09
	R2	21.3	14,334,770.9	1,142,693.3	758,262.8	8.86	4.92	2.85	1.80	3.10

Table 5: Construction heuristic results – duration costs – $n = 1000$ customers.

						Average CPU (s)			Speed-up o. NAIVE	
		#routes	#mov.	#PCfeasmov.	#feasmov.	NAIVE	TREE	F/B	TREE	F/B
TD0	C1	101.3	3,423,521.2	270,889.6	270,889.6	1.90	1.82	1.69	1.05	1.13
	RC1	106.0	3,533,413.7	307,888.9	307,888.9	2.01	1.86	1.72	1.08	1.17
	R1	103.9	3,553,110.7	477,005.9	477,005.9	2.26	2.05	1.83	1.10	1.24
	C2	35.1	8,947,100.9	540,249.8	540,249.8	3.18	2.29	1.94	1.39	1.64
	RC2	34.7	12,496,541.9	708,577.0	708,577.0	4.72	2.65	2.13	1.78	2.21
	R2	29.6	13,075,602.8	1,148,350.8	1,148,350.8	6.79	3.30	2.41	2.06	2.82
TD1	C1	102.9	3,451,578.7	407,555.8	355,700.2	2.82	2.70	2.24	1.04	1.26
	RC1	109.1	3,524,992.4	475,308.3	404,702.8	3.46	3.00	2.35	1.15	1.47
	R1	116.6	3,529,151.9	661,467.3	614,328.0	4.28	3.58	2.67	1.19	1.61
	C2	34.8	9,038,582.8	858,311.5	763,742.0	12.49	5.88	3.76	2.12	3.32
	RC2	36.6	12,200,695.9	1,208,980.3	1,051,252.1	49.15	13.71	6.34	3.59	7.76
	R2	35.5	12,493,647.4	1,488,960.8	1,411,922.9	74.81	17.10	8.54	4.38	8.76
TD2	C1	109.5	3,456,519.0	478,188.4	414,159.1	3.10	3.00	2.43	1.03	1.27
	RC1	116.1	3,517,931.8	508,909.9	405,680.0	3.62	3.07	2.43	1.18	1.49
	R1	122.0	3,537,531.0	667,778.4	612,971.7	4.56	3.71	2.78	1.23	1.64
	C2	35.2	9,107,428.3	1,094,623.0	978,532.8	13.94	6.51	4.12	2.14	3.38
	RC2	38.1	11,914,694.5	1,569,033.9	1,317,501.2	70.30	19.76	8.86	3.56	7.93
	R2	36.3	12,663,590.7	1,386,494.6	1,308,252.1	48.13	13.41	6.49	3.59	7.42
TD3	C1	114.6	3,452,527.3	552,267.4	465,432.2	3.39	3.21	2.51	1.05	1.35
	RC1	119.7	3,500,667.4	563,992.7	430,201.7	3.81	3.25	2.51	1.17	1.52
	R1	129.9	3,520,679.9	698,052.6	622,257.3	4.78	3.83	2.81	1.25	1.70
	C2	35.2	9,060,139.2	1,345,638.7	1,200,492.9	17.03	8.22	4.86	2.07	3.51
	RC2	40.0	11,909,922.2	1,658,617.5	1,317,798.6	72.03	19.96	8.92	3.61	8.07
	R2	37.3	12,605,706.6	1,538,136.5	1,444,501.2	72.90	16.17	8.20	4.51	8.89

Table 6: Lexicographic 8-exchange results – $n = 1000$ customers.

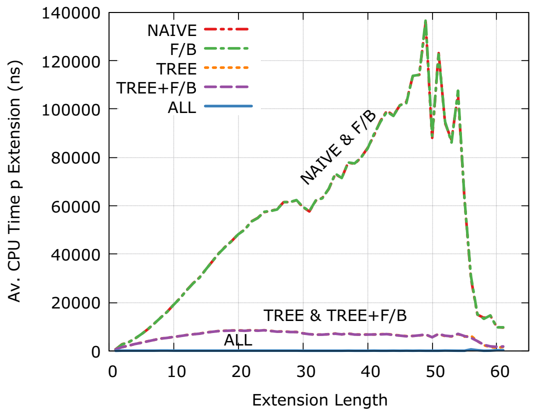
						Average CPU (s)					Speed-up over NAIVE			
		#It	#mov.	#PCfeasmov.	#feasmov.	NAIVE	TREE	F/B	TREE +F/B	ALL	TREE	F/B	TREE +F/B	ALL
TD0	C1	243.4	133,245,009.7	33,557.8	11,426.7	6.31	6.31	6.28	6.30	6.33	1.00	1.00	1.00	1.00
	RC1	488.8	309,583,214.7	307,664.2	123,642.0	13.38	13.14	12.96	12.93	12.87	1.02	1.03	1.03	1.04
	R1	418.6	265,539,145.5	618,157.8	174,454.5	12.05	11.65	11.39	11.27	11.08	1.03	1.06	1.07	1.09
	C2	142.8	403,395,382.2	296,601.1	18,116.4	22.19	21.78	21.60	21.58	21.82	1.02	1.03	1.03	1.02
	RC2	178.5	845,004,596.0	22,548,303.1	1,962,978.5	105.17	71.06	58.88	56.03	50.71	1.48	1.79	1.88	2.07
	R2	181.3	890,270,828.2	36,885,040.5	2,410,307.5	146.60	90.38	71.39	66.73	57.73	1.62	2.05	2.20	2.54
TD1	C1	265.2	151,428,561.3	1,100,200.1	150,226.1	12.14	11.33	10.34	9.87	8.97	1.07	1.17	1.23	1.35
	RC1	534.4	344,519,537.6	1,229,367.2	355,145.9	20.27	18.27	17.45	16.66	15.66	1.11	1.16	1.22	1.29
	R1	481.8	308,158,527.9	2,001,607.0	484,879.3	22.75	19.57	18.57	16.91	15.07	1.16	1.23	1.35	1.51
	C2	178.7	509,306,192.6	18,963,511.9	461,662.9	124.91	92.88	69.79	64.31	49.94	1.34	1.79	1.94	2.50
	RC2	205.4	987,288,907.5	48,963,091.0	4,822,667.3	929.13	386.68	257.51	235.82	170.16	2.40	3.61	3.94	5.46
	R2	218.1	1,089,008,156.0	57,897,750.6	3,173,019.5	594.36	353.03	222.63	191.10	115.50	1.68	2.67	3.11	5.15
TD2	C1	289.2	165,125,066.9	1,794,314.5	268,090.6	15.97	14.55	13.03	12.21	10.73	1.10	1.23	1.31	1.49
	RC1	542.7	354,288,954.3	1,879,741.2	534,412.2	24.57	21.61	20.27	19.02	17.38	1.14	1.21	1.29	1.41
	R1	574.2	367,765,351.5	2,903,340.6	857,239.0	31.10	26.47	24.32	21.99	19.19	1.17	1.28	1.41	1.62
	C2	188.3	563,514,893.2	33,443,587.0	609,260.5	211.88	146.74	106.99	97.05	70.36	1.44	1.98	2.18	3.01
	RC2	223.9	1,105,241,530.7	46,213,980.3	5,742,931.5	513.78	348.56	201.33	178.53	116.17	1.47	2.55	2.88	4.42
	R2	259.2	1,314,357,671.3	83,270,404.5	7,135,950.5	916.22	515.75	318.28	276.55	170.12	1.78	2.88	3.31	5.39
TD3	C1	298.5	174,860,286.8	2,515,959.7	346,823.2	19.74	17.65	15.63	14.35	12.17	1.12	1.26	1.38	1.62
	RC1	590.0	383,723,779.3	2,751,181.5	810,596.5	29.75	25.84	23.72	22.05	19.73	1.15	1.25	1.35	1.51
	R1	603.0	393,329,002.4	3,526,909.0	1,061,401.1	36.04	30.06	27.52	24.61	21.10	1.20	1.31	1.46	1.71
	C2	184.1	562,132,609.0	52,675,904.2	917,470.8	312.82	219.27	154.38	138.38	93.75	1.43	2.03	2.26	3.34
	RC2	230.9	1,121,656,354.6	57,124,211.0	6,800,568.9	498.24	395.30	215.88	191.79	120.50	1.26	2.31	2.60	4.13
	R2	280.0	1,437,934,819.2	116,147,438.8	9,984,865.4	1,213.36	763.68	445.26	383.56	221.46	1.59	2.73	3.16	5.48

One cause of the speed-up of the TREE, TREE+F/B and ALL methods over NAIVE and F/B during lexicographic exchange is the decreased CPU time needed to update the middle segment ready time functions with a number of new customers. This happens in Lexicographic Search when between two feasible moves a number of moves are found to be infeasible by the pre-checks and the middle segment ready time function is updated in a lazy fashion. To illustrate the differences in middle segment extension times, Figure 3(a) shows the average CPU times needed in nanoseconds per middle segment ready time function extension by a certain number of customers during the first iteration of M-exchange on instance R2_10_04 with speed-profile TD3. Also the total CPU times in seconds spent on middle segment ready time function extension of a certain length during the iteration and the cumulative total CPU times are shown in Figure 3(b) and Figure 3(c), respectively. Recall that both NAIVE and F/B methods extend middle segment ready time functions by successive forward composition, resulting in a quadratic time complexity in number of extension customers (see Lemma 5), while the TREE and TREE+F/B methods only need linear time (see Theorem 8) and ALL no time at all. Figure 3(a) shows these differences empirically, especially for the lower extension lengths. The TREE method even seems to perform better on average than its worse-case predicted linear time. For large extension lengths, the average CPU times for most methods become noisy due to the low number of such large extensions observed in our experiment.

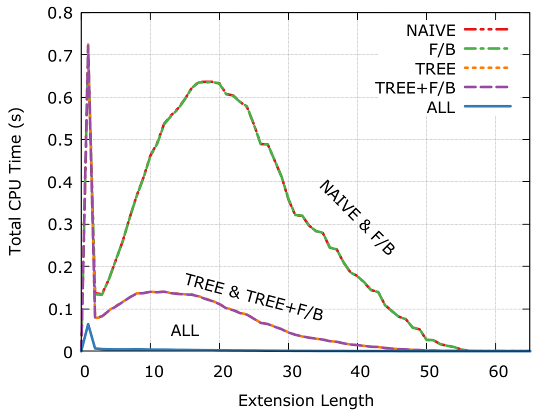
The results of the segment extension measurements as illustrated in Figure 3 suggest that the speed-up of TREE compared to F/B will increase as the maximum segment length k increases. Table 7 shows the results of the k-exchange runs with maximum subsequence lengths $k = 2, 4, 8, 16, 32, M$. Here, only the long-route instance groups C2, RC2, and R2 are shown, because the routes of the other groups are too short, on average 10 customers, to apply higher maximum exchange subsequence lengths. Notice that in this table the speed-up is given compared to F/B to illustrate the additional benefits of TREE+F/B and ALL over F/B when subsequence lengths k are large. For large k, TREE+F/B is able to offer an additional speed-up of up to 1.56 on top of F/B, while ALL offers a speed-up of up to 2.83 over F/B (1.87 over TREE+F/B) at the cost of more memory needed.

8.4. Non-lexicographic Exchange Experiments

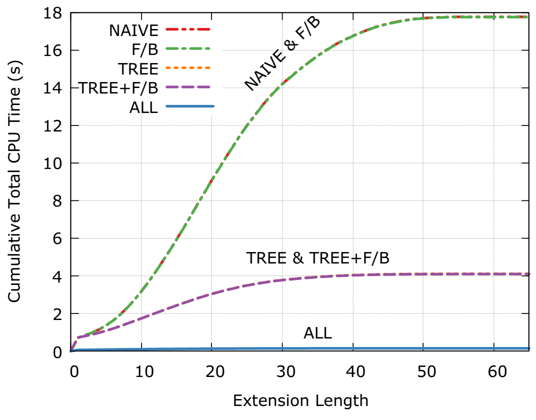
We have seen that the addition of ready time function trees in the TREE+F/B method compared to only forward/backward ready time functions in the F/B method give a speed-up during lexicographic exchange.



(a) Average CPU time per extension (ns).



(b) Total extension time (s).



(c) Cumulative total extension time (s).

Figure 3: Middle segment extension CPU times during the first iteration of M-exchange on instance TD3 R2_10_04.

To show the benefits of the ready time function tree in non-lexicographic neighborhood search, we measure CPU times of executing single iterations of a fixed k-exchange neighborhood. In such a neighborhood, only exchanges of two subsequences with exactly k customers are evaluated. Notice that such neighborhoods cannot be searched lexicographically (or do not benefit from this).

Table 8 shows the total computation times, averaged over the instance groups, needed to do a single iteration of fixed k-exchanges from $k = 1$ up to $k = M$ and the corresponding speed-up over the NAIVE method. Only the long-route large-TW instance groups, C2, RC2 and R2, are shown, because the other instances had too short routes. Both the TREE and TREE+F/B perform better than the F/B method. This illustrates the improved non-lexicographic move complexity of the TREE-based methods over the F/B method.

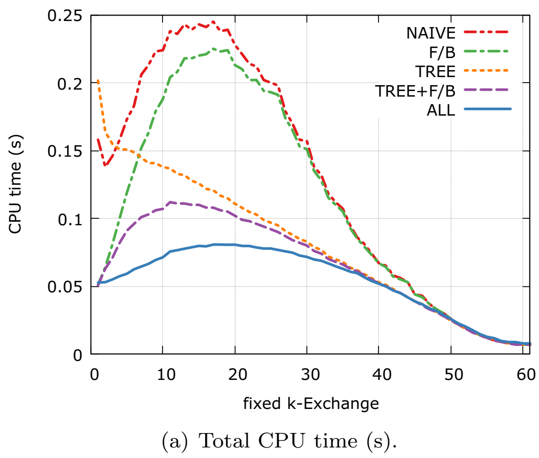
Figure 4(a) shows the CPU time for running one iteration of fixed k-exchange for varying k in the construction heuristic solution of instance R2_10_04 with speed-profile TD3. Also the cumulative total CPU times are shown in Figure 4(b). The figures show that the total CPU times of TREE and TREE+F/B follow those of ALL rather than following F/B when increasing k. Although the F/B method provides a high speed-up for low k, the TREE-based methods provide a high speed-up for high subsequence length k. This is in line with the non-lexicographic move complexities presented in Tables 1 and 2. The figures also illustrate the benefits of combining the TREE and F/B methods into TREE+F/B to decrease computation times.

Table 7: Lexicographic k-exchanges results – $n = 1000$ customers – Speed-profile TD3.

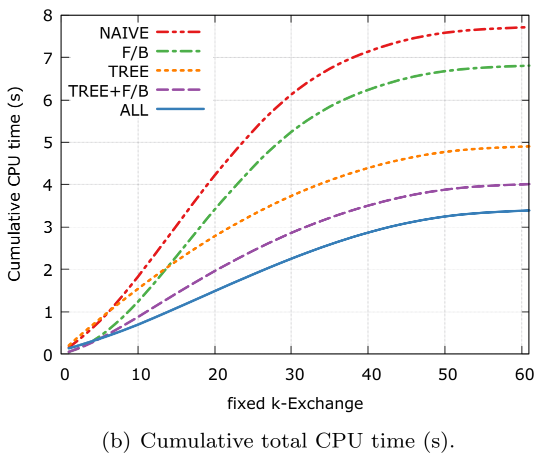
						Average CPU (s)			Sp.-up o. F/B	
		#It	#mov.	#PCfeasmov.	#feasmov.	F/B	TREE +F/B	ALL	TREE +F/B	ALL
$k = 2$	C2	147.4	34,413,792.5	4,556,501.6	131,673.7	7.93	7.81	6.53	1.01	1.21
	RC2	160.6	56,255,515.9	4,302,823.9	813,716.1	12.85	12.79	11.72	1.00	1.10
	R2	211.2	77,322,050.0	8,939,513.8	1,086,952.8	17.55	17.45	14.77	1.01	1.19
$k = 4$	C2	167.9	147,397,603.0	17,235,212.8	417,412.0	34.56	32.90	24.15	1.05	1.43
	RC2	194.5	263,205,979.9	16,446,865.9	2,839,082.7	50.27	48.39	37.62	1.04	1.34
	R2	258.3	360,998,852.4	34,209,933.8	4,668,740.8	93.75	88.81	65.54	1.06	1.43
$k = 8$	C2	184.1	562,132,609.0	52,675,904.2	917,470.4	154.38	138.38	93.75	1.12	1.65
	RC2	230.9	1,121,656,354.6	57,124,211.0	6,800,568.9	215.88	191.79	120.50	1.13	1.79
	R2	280.0	1,437,934,819.2	116,147,438.8	9,984,865.4	445.26	383.56	221.46	1.16	2.01
$k = 16$	C2	220.9	1,962,306,197.3	126,298,069.5	1,914,591.0	315.06	257.69	173.43	1.22	1.82
	RC2	256.1	4,025,788,272.9	203,148,876.0	24,781,583.0	801.11	704.17	469.89	1.14	1.70
	R2	306.7	5,312,619,679.9	269,700,418.7	10,710,454.3	906.12	681.79	346.35	1.33	2.62
$k = 32$	C2	218.2	3,528,300,535.3	171,578,568.4	2,418,781.9	486.21	355.27	243.36	1.37	2.00
	RC2	243.5	9,438,707,760.0	447,270,487.9	46,460,316.6	1,588.52	1,309.84	872.85	1.21	1.82
	R2	284.2	12,481,222,179.8	525,720,866.4	26,408,131.0	2,643.13	1,721.28	924.76	1.54	2.86
$k = M$	C2	210.2	3,447,702,250.0	174,267,323.3	2,159,624.5	475.70	345.98	234.16	1.37	2.03
	RC2	273.4	14,696,107,214.8	484,354,526.4	48,907,276.7	1,740.77	1,340.06	876.40	1.30	1.99
	R2	276.3	17,005,163,425.9	640,998,884.0	32,736,498.9	2,599.30	1,670.21	918.00	1.56	2.83

Table 8: Non-lexicographic fixed k-exchanges results – first iteration – $n = 1000$ customers.

		Average CPU (s)					Speed-up over NAIVE			
		NAIVE	TREE	F/B	TREE +F/B	ALL	TREE	F/B	TREE +F/B	ALL
TD3	C2	9.80	5.79	8.10	4.54	3.30	1.69	1.21	2.16	2.97
	RC2	5.61	4.19	4.87	3.73	3.48	1.34	1.15	1.51	1.61
	R2	6.41	4.91	5.58	4.49	4.34	1.30	1.15	1.43	1.48



(a) Total CPU time (s).



(b) Cumulative total CPU time (s).

Figure 4: Non-lexicographic fixed k-exchange results – CPU-times in seconds for the first iteration of instance TD3 R2_10_04 for each fixed subsequence length k.

8.5. Memory usage

The investigated pre-calculation methods TREE, F/B, TREE+F/B and ALL have each shown to speed-up insertion and exchange neighborhood search running times over NAIVE. However, this speed-up comes at a cost of increased memory usage, as shown in worst-case complexities in Tables 1 and 2. To verify this empirically, we measured the additional memory used by each method during the lexicographic 8-exchange runs reported in Table 6. Table 9 shows the maximum memory, in number of breakpoints, that was used by each method during these 8-exchange runs, averaged over the ten instances per instance group. In this

Table 9: Additional memory usage during 8-exchange.

	Additional Memory Needed (#BreakPoints)				
	NAIVE	TREE	F/B	TREE+F/B	ALL
TD0 C1	–	4,073.0	5,831.4	9,895.1	13,393.9
RC1	–	3,993.2	5,202.7	9,176.3	12,116.4
R1	–	3,869.4	4,737.8	8,596.5	10,647.6
C2	–	4,266.2	5,685.6	9,946.8	32,592.0
RC2	–	4,176.1	4,667.6	8,836.4	37,308.0
R2	–	4,155.7	4,543.5	8,689.4	36,737.4
TD3 C1	–	9,589.1	11,361.3	20,886.0	29,000.3
RC1	–	8,277.3	8,483.4	16,705.4	23,448.5
R1	–	10,152.9	8,952.0	18,988.7	26,720.4
C2	–	9,745.3	10,270.0	19,917.6	76,340.1
RC2	–	8,752.9	9,292.9	18,001.3	126,385.9
R2	–	12,004.9	10,340.9	21,741.7	207,990.5
TD3 usa13509	–	273,460.0	1,349,116.0	1,622,576.0	61,350,064.0

table, we only show results of the non-time dependent speed-profile TD0 and speed-profile TD3. While the TREE and F/B methods seem to require an equal amount of memory and TREE+F/B twice that amount, the ALL method needs up to 10 times the amount of memory used by TREE+F/B and up to 20 times the amount used by either TREE and F/B in case of speed-profile TD3. Note that the speed-up of ALL on the 1000 customer instances with $k = 8$ over the TREE+F/B method is up to 1.75 times and for $k = M$ up to 1.86 times (see Table 7), while the required memory is much more. Furthermore, to emperically illustrate the memory complexities presented in Tables 1 and 2, we include the memory usage for a much larger instance, the TD3 usa13509 instance. We created this instance since by our knowlegde no VRPTW instances with more than 1000 customers are commonly used currently in the literature. This instance has $n = 13508$ customers based on the well-known TSP-LIB [21] instance *usa13509*, using additionally vertex 1 as depot, 150 vehicles with capacity $Q = 250$, time horizon $b_d = 1000000$, customers each require 1 quantity and have a time window either $[0, b_d/2]$, $[b_d/4, 3b_d/4]$, or $[b_d/2, b_d]$, and using speed-profile TD3. On this instance the 8-exchange is run for one iteration after the cheapest insertion construction heuristic. The lower complexity of TREE compared to F/B can be seen empirically as the latter requires almost 5 times more memory. The ALL method requires almost 38 times more memory than the TREE+F/B method. In all experiments, as indicated by the complexities, the memory requirement of ALL grows much faster than the other methods. This would make ALL an unsuitable method when memory is limited, for instance when using fast CPU caches or parallel processes on GPUs.

9. Other applications

In this section, we briefly illustrate the general applicability of the presented methods by considering two other applications of the presented speed-up methods.

9.1. Multiple Time Windows

The presented methods in this paper can easily be adapted to solve the TDVPRTW with route duration constraints and objective and with additionally that customers have multiple time windows. For this, only the time window ready time functions of Definition 1 need to be modified. For a customer $i \in \mathcal{V}^{\mathrm{C}}$ with a number of w time windows $\left[a_i^1, b_i^1\right], \left[a_i^2, b_i^2\right], \ldots, [a_i^w, b_i^w]$, the time window ready time function θ_i is given by:

$$\theta_i(t) := \begin{cases} a_i^1 + s_i & \text{if } t < a_i^1, \\ t_i + s_i & \text{if } t \in \left[a_i^1, b_i^1\right], \\ a_i^2 + s_i & \text{if } b_i^1 < t < a_i^2, \\ t_i + s_i & \text{if } t \in \left[a_i^2, b_i^2\right], \\ \vdots & \vdots \\ a_i^m + s_i & \text{if } b_i^{w-1} < t < a_i^w, \\ t_i + s_i & \text{if } t \in [a_i^w, b_i^w]. \end{cases} \tag{13}$$

This function has $\mathcal{O}(w)$ number of breakpoints. Figure 5 shows an example of a time window ready time function in case of a customer with $w = 2$ time windows. This multiple time window ready time function has very similar properties as the arrival time functions. Note that this function is not continuous but lower semi-continuous, however function composition can be shown to preserve semi-lower continuity. Therefore, all of the presented complexity results in Tables 1 and 2 only the p changes to $p + w$ or, equivalently, to $\max\{p, w\}$.

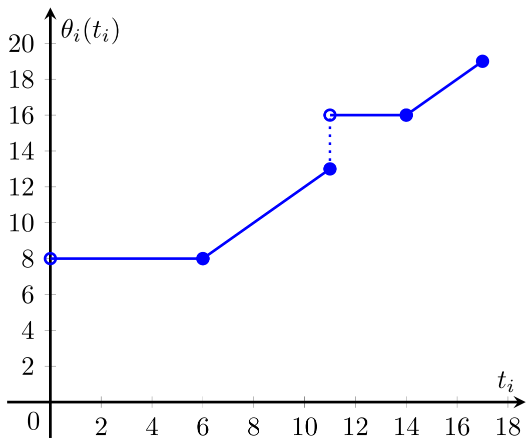


Figure 5: An example of a multiple time window ready time function $\theta_i(t_i)$ for TWs $[a_i^1, b_i^1] = [6, 11]$, $[a_i^2, b_i^2] = [14, 17]$ and $s_i = 2$.

9.2. Pre-checks

Although the presented speed-up methods enable fast move evaluations for our TDVRPTW, they can also be used as pre-checks for move evaluations of more difficult routing problems. For instance, consider the time-dependent vehicle routing problem with driver legislations (see for instance Kok et al. [19]), which includes additional constraints such as driving- and working time breaks a driver needs to take during his shift. Since our problem with route duration constraints is a relaxation of this problem, the presented methods can serve as pre-check before conducting the more time-consuming exact feasibility check. Notice that pre-checks based on our TDVRPTW are generally stronger for the more difficult problem than pre-checks based on the classical vehicle routing problem with duration constraints, which can result in speed-ups.

10. Conclusion

In this paper, we investigate the time-dependent vehicle routing problem with time windows, route duration constraints and duration minimization, and propose methods to speed-up common Neighborhood Search based heuristics by decreasing move evaluation CPU times. The inclusion of both time-dependent travel times and route duration constraints and objective are important to model real-world problem features such as road congestion and driver's maximum working shift duration, but increase the neighborhood move evaluation complexity.

The analysis of ready time functions leads to promising preprocessing methods to increase the efficiency of move evaluations. We presented the F/B method which stores forward and backward ready time functions in memory, which reduces the move evaluation complexity from quadratic in number of customers to linear. Empirical results on 1000 customer benchmark instances show speed-ups of up to 8.89 times during construction and up to 3.61 times during exchange with limited subsequence length. Speed-ups increase significantly as the number of customers in a route or as the max exchange subsequence length increase.

To decrease the exchange move evaluation times further, we developed a new tree-based data structure of ready time functions. It allows move evaluation complexity to remain linear even when the search is conducted using a non-lexicographical order. We presented a method using only the tree-based data structure, TREE, and a combined method, TREE+F/B. Empirically, benefits are also observed in Lexicographic Searches, where speed-ups of the TREE+F/B method of up to 1.56 are achieved on top of the F/B method. This speed-up is attained without increasing the order of memory required. The most speed-up was observed using a method storing all partial ready time functions, ALL, up to 1.87 on top of the TREE+F/B method, while the required memory increased cubically.

Finally, we presented two other applications of the speed-up methods including Multiple Time Windows. These illustrate the general applicability of the investigated speed-up techniques.

Acknowlegdements

This research was funded by NWO, the Netherlands Organisation for Scientific Research, as part of the multi-annual research programme on Sustainable Logistics.

References

[1] Balseiro, S., Loiseau, I., Ramonet, J., 2011. An ant colony algorithm hybridized with insertion heuristics for the time dependent vehicle routing problem with time windows. Computers & Operations Research 38 (6), 954 – 966.

[2] Campbell, A. M., Savelsbergh, M. W. P., 2004. Efficient insertion heuristics for vehicle routing and scheduling problems. Transportation Science 38 (3), 369–378.

[3] Dabia, S., Ropke, S., van Woensel, T., De Kok, T., 2013. Branch and price for the time-dependent vehicle routing problem with time windows. Transportation Science 47 (3), 380–396.

[4] de Berg, M., Cheong, O., van Kreveld, M., Overmars, M., 2008. Chapter 5: Orthogonal range searching. In: Computational Geometry: Algorithms and Applications. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 95–120.

[5] Desaulniers, G., Madsen, O. B., Ropke, S., 2014. Chapter 5: The Vehicle Routing Problem with Time Windows. In: Toth, P., Vigo, D. (Eds.), Vehicle Routing: Problems, Methods, and Applications, 2nd Edition. Vol. 18 of MOS-SIAM Series on Optimization. SIAM - Society for Industrial and Applied Mathematics, Philadelphia, Ch. 5, pp. 119–159.

[6] Donati, A. V., Montemanni, R., Casagrande, N., Rizzoli, A. E., Gambardella, L. M., 2008. Time dependent vehicle routing problem with a multi ant colony system. European Journal of Operational Research 185 (3), 1174 – 1191.

[7] Figliozzi, M. A., 2012. The time dependent vehicle routing problem with time windows: Benchmark problems, an efficient solution algorithm, and solution characteristics. Transportation Research Part E: Logistics and Transportation Review 48 (3), 616 – 636.

[8] Garcia, A., Vansteenwegen, P., Arbelaitz, O., Souffriau, W., Linaza, M. T., 2013. Integrating public transportation in personalised electronic tourist guides. Computers & Operations Research 40 (3), 758 – 774.

[9] Gavalas, D., Konstantopoulos, C., Mastakas, K., Pantziou, G., Vathis, N., 2015. Heuristics for the time dependent team orienteering problem: Application to tourist route planning. Computers & Operations Research 62, 36 – 50.

[10] Gehring, H., Homberger, J., 2001. A parallel two-phase metaheuristic for routing problems with time windows. Asia - Pacific Journal of Operational Research 18 (1), 35–47.

[11] Gendreau, M., Ghiani, G., Guerriero, E., 2015. Time-dependent routing problems: A review. Computers & Operations Research 64, 189 – 197.

[12] Ghiani, G., Guerriero, E., 2014. A Note on the Ichoua, Gendreau, and Potvin (2003) Travel Time Model. Transportation Science 48 (3), 458–462.

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

[14] Hashimoto, H., Yagiura, M., Ibaraki, T., 2008. An iterated local search algorithm for the time-dependent vehicle routing problem with time windows. Discrete Optimization 5 (2), 434 – 456.

[15] Ichoua, S., Gendreau, M., Potvin, J.-Y., 2003. Vehicle dispatching with time-dependent travel times. European Journal of Operational Research 144 (2), 379–396.

[16] Irnich, S., 2008. A unified modeling and solution framework for vehicle routing and local search-based metaheuristics. INFORMS Journal on Computing 20 (2), 270–287.

[17] Kindervater, G. A., Savelsbergh, M. W., 1997. Vehicle routing: Handling edge exchanges. In: Aarts, E., Lenstra, J. K. (Eds.), Local Search in Combinatorial Optimization, 1st Edition. John Wiley & Sons, Inc., New York, NY, USA, pp. 337–360.

[18] Kok, A., Hans, E., Schutten, J., 2011. Optimizing departure times in vehicle routes. European Journal of Operational Research 210 (3), 579 – 587.

[19] Kok, A. L., Hans, E. W., Schutten, J. M. J., Zijm, W. H. M., 2010. A dynamic programming heuristic for vehicle routing with time-dependent travel times and required breaks. Flexible Services and Manufacturing Journal 22 (1), 83–108.

[20] Labadie, N., Prins, C., Prodhon, C., 2016. Metaheuristics generating a sequence of solutions. In: Metaheuristics for Vehicle Routing Problems, 1st Edition. John Wiley & Sons, Inc., Hoboken, NJ, USA, pp. 39–75.

[21] Reinelt, G., 1991. TSPLIB–A Traveling Salesman Problem Library. ORSA Journal on Computing 3 (4), 376–384.

[22] Savelsbergh, M. W. P., 1992. The vehicle routing problem with time windows: Minimizing route duration. ORSA Journal on Computing 4 (2), 146–154.

[23] Verbeeck, C., Srensen, K., Aghezzaf, E.-H., Vansteenwegen, P., 2014. A fast solution method for the time-dependent orienteering problem. European Journal of Operational Research 236 (2), 419 – 432.

[24] Vidal, T., Crainic, T. G., Gendreau, M., Prins, C., 2013. Heuristics for multi-attribute vehicle routing problems: A survey and synthesis. European Journal of Operational Research 231 (1), 1 – 21.

[25] Vidal, T., Crainic, T. G., Gendreau, M., Prins, C., 2015. Timing problems and algorithms: Time decisions for sequences of activities. Networks 65 (2), 102–128.

[26] Zachariadis, E. E., Kiranoudis, C. T., 2010. A strategy for reducing the computational complexity of local search-based methods for the vehicle routing problem. Computers & Operations Research 37 (12), 2089 – 2105.