Algorithmica (2022) 84:2597–2621
https://doi.org/10.1007/s00453-022-00974-4

Approximation Algorithms for Replenishment Problems with Fixed Turnover Times

Thomas Bosman[1] · Martijn van Ee[2] · Yang Jiao[3] · Alberto Marchetti-Spaccamela[4,5] · R. Ravi[6] · Leen Stougie[5,7,8]

Received: 27 December 2019 / Accepted: 14 April 2022 / Published online: 16 May 2022

Abstract
We introduce and study a class of optimization problems we call replenishment problems with fixed turnover times: a very natural model that has received little attention in the literature. Clients with capacity for storing a certain commodity are located at various places; at each client the commodity depletes within a certain time, the turnover time, which is constant but can vary between locations. Clients should never run empty. The natural feature that makes this problem interesting is that we may schedule a replenishment (well) before a client becomes empty, but then the next replenishment will be due earlier also. This added workload needs to be balanced against the cost of routing vehicles to do the replenishments. In this paper, we focus on the aspect of minimizing routing costs. However, the framework of recurring tasks, in which the next job of a task must be done within a fixed amount of time after the previous one is much more general and gives an adequate model for many practical situations. Note that our problem has an infinite time horizon. However, it can be fully characterized by a compact input, containing only the location of each client and a turnover time. This makes determining its computational complexity highly challenging and indeed it remains essentially unresolved. We study the problem for two objectives: MIN–AVG minimizes the average tour cost and MIN–MAX minimizes the maximum tour cost over all days. For MIN–MAX we derive a logarithmic factor approximation for the problem on general metrics and a 6-approximation for the problem on trees, for which we have a proof of NP-hardness. For MIN–AVG we present a logarithmic factor approximation on general metrics, a 2-approximation for trees, and

A preliminary version of this article appeared in [7]. The research of YJ and RR was supported in part by the U. S. National Science Foundation under award numbers CCF-1527032 and CCF-1655442, and the U. S. Office of Naval Research under award number N00014-18-1-2099. The research of LS was supported by the Netherlands Organisation for Scientific Research (NWO) through the Gravitation Programme Networks (024.002.003).

✉ Martijn van Ee
m.v.ee.01@mindef.nl

Extended author information available on the last page of the article

a pseudopolynomial time algorithm for the line. Many intriguing problems remain open.

Keywords Approximation algorithm · Inventory routing · Replenishment problem · Periodic maintenance · Pinwheel scheduling

1 Introduction

Imagine the following particular inventory-routing problem. A set of automatic vendor machines are spread over a country or a city. They have a certain turnover time: the number of days in which a full machine will be sold out. Replenishment is done by vehicles. Let us assume that turnover times are machine/location dependent but not time dependent, and that it is highly undesirable to have an empty machine. However, the holding costs of a machine are negligible, so that we will always fill a machine to capacity. There is nothing against replenishing a machine before it has become empty, but then the next replenishment will be due earlier as well. That is, the deadline of the next replenishment is always within one turnover time after the last replenishment. Equivalently, in any consecutive number of days equal to the turnover time, at least one replenishment has to take place. Replenishing a machine earlier to combine it with the replenishment of another machine that is due earlier may lead to cost savings in routing of the vehicles. The feature that makes this problem special w.r.t. those in the existing literature is that it can be compactly modeled by only specifying for every machine its location and its turnover time. The feature is very natural but has hardly been studied in the existing literature. There are intriguing basic open complexity questions, and some highly non-trivial results.

The motivation for studying this problem comes directly from a business project for the replenishment of ATMs in the Netherlands, in which some of the co-authors are involved. Of course the real-life ATM replenishment problem is not as stylized as described above; the turnover time is not strictly the same over time but subject to variability, there are restrictions on the routes for the vehicles, etc. But the feature that is least understood in the ATM-problem is exactly the problem of how to deal with the trade-off between replenishing an ATM earlier than its due date, leading to a higher frequency of replenishments, and the savings on vehicle routing costs.

Formally, an instance of the problem that we study in this paper, which we name the REPLENISHMENT PROBLEM WITH FIXED TURNOVER TIMES (RFTT), consists of a pair (G, τ), where $G = (V \cup \{s\}, E, c)$ is a weighted graph with a designated root vertex s, costs on the edges $c : E \to \mathbb{R}_+$, and turnover times $\tau \in \mathbb{N}^{|V|}$, indicating that $v_j \in V$ should be visited at least once in every interval of τ_j days. Throughout the paper we use day as our standard time unit.

A solution consists, for each day k, of a tour T_k in G starting in and returning to the root s and visiting a subset of the vertices $J_k \subseteq V$. It is feasible if $v_j \in \bigcup_{k=t+1}^{t+\tau_j} J_k, \forall t$ and $\forall v_j \in V$. We will focus on solutions that repeat themselves after a finite amount of time, that is, in which $(T_k, ..., T_{k+\ell}) = (T_{k+\ell+1}, ..., T_{k+2\ell})$ for some ℓ, and all k. Since all turnover times are finite, this is no real restriction.

Since ℓ, the length of the schedule, can be exponential in the size of the graph, we need to restrict our solution space in order to obtain polynomial time algorithms. We choose to consider only periodic solutions, which have the property that the time between each two consecutive visits to a vertex is a fixed quantity. To efficiently represent periodic solutions that repeat after time $\hat{t}$ we use an array that for each vertex $v \in V$ stores two positive values $h(v)$ and $p(v)$; $h(v) + \hat{t}$ denotes the first day v is visited after $\hat{t}$, while $p(v)$ stores the period of consecutive visits to v. Therefore for $t > \hat{t}$ we have that v is visited at days $h(v) + \hat{t} + kp(v)$, $k = 0, 1, 2, \ldots$. The above data structure easily allows to represent the set of vertices to be visited for each day t, $t > \hat{t}$; in fact on day t we have to visit vertices such that $t - h(v) - \hat{t}$ is congruent to $p(v)$. Algorithms presented in the sequel directly compute or will allow to easily compute values $h(v)$ and $p(v)$ for each vertex v in the case $\hat{t} = 0$.

We consider two versions of RFTT. In the first version, called MIN–AVG RFTT, the goal is to find a feasible solution that minimizes the average tour cost. Note that this is well-defined since we consider solutions that repeat themselves after a finite amount of time. In MIN–MAX RFTT we aim to find a feasible solution that minimizes the maximum tour cost over all days.

We emphasize that the particular feature of this model, that visits to vertices recur and need to be carried out within each vertex-specific consecutive time interval, occurs naturally in many problem settings. It allows any job of a recurring task to be done before its deadline, but then the next job of the task comes earlier and hence its deadline. This is a feature that, despite its natural applicability, has hardly been studied in the literature from a theoretical point of view.

Related work.

Our problem can be seen as a special case of the INVENTORY ROUTING PROBLEM (IRP) [12]. Here, clients have their own storage with a certain capacity and for each day a demand is specified. The clients pay holding cost over their inventory. However, omitting inventory cost, we can interpret our problem as such an inventory routing problem in which the demand at any given location is the same every day, leading to a very small input description of our problem consisting only of a location and a turnover time (storage capacity divided by daily demand), which makes it incomparable to the inventory routing problem from a complexity point of view. Indeed it is unclear if the decision version of our problem is in NP or in co-NP.

Another closely related problem is the PERIODIC LATENCY PROBLEM [13], which features the recurring visits requirement of RFTT. We are given recurrence length q_i for each client i and travel distances between clients. Client i is considered *served* if it is visited every q_i days. The server does not return to the root at the end of each day, but keeps moving continuously between clients at uniform speed. Another difference between PERIODIC LATENCY and RFTT is the objective function. Coene et al. [13] study two versions of the problem: one that maximizes the number of served clients by one server, and one that minimizes the number of servers needed to serve all clients. They resolve the complexity of these problems on lines, circles, stars, trees, and general metrics. In the PERIODIC LATENCY PROBLEM, each client is assigned to one server. The case in which a client can be served by multiple servers is studied in patrolling problems [11, 14].

A problem that does share the compact input size and is in fact a very special case of our problem is known under the guise of Pinwheel Scheduling. It has been introduced to model the scheduling of a ground station to receive information from a set of satellites without data loss. This corresponds to the version of our problem where no more than one vertex can be replenished per day and all distances to the root are the same; the interesting question here is if there exists a feasible schedule for replenishing the vertices. Formally, a set of jobs $\{1, ..., n\}$ with periods $p_1, ..., p_n$ is given, and the question is whether there exists a schedule $\sigma : \mathbb{N} \to \{1, .., n\}$ such that $j \in \bigcup_{k=t+1}^{t+p_j} \sigma_k$, $\forall t \geq 0$ and $\forall j$.

Pinwheel Scheduling was introduced by Holte et al. [24], who showed that it is contained in PSPACE. The problem is in NP if the schedule σ is restricted to one in which for each job the time between two consecutive executions remains constant throughout the schedule, a so-called periodic schedule. In particular this holds for instances with density $\rho = \sum_j 1/p_j = 1$ [24]. They also observed that the problem is easily solvable when $\rho \leq 1$ and the periods are harmonic, i.e. p_i is a divisor of p_j or vice versa for all i and j. As a corollary, every instance with $\rho \leq \frac{1}{2}$ is feasible.

Chan and Chin [10] improved the latter result by giving an algorithm that produces a feasible schedule for Pinwheel Scheduling whenever $\rho \leq \frac{2}{3}$. In [9], they improved this factor to $\frac{7}{10}$. Later, Fishburn and Lagarias [20] showed that every instance with $\rho \leq \frac{3}{4}$ has a feasible schedule. All these papers work towards the conjecture that there is a feasible schedule if $\rho \leq \frac{5}{6}$. That this bound is tight can be seen by the instance with $p_1 = 2$, $p_2 = 3$ and $p_3 = M$, with M large. This instance cannot be scheduled, but has a density of $\frac{5}{6} + \frac{1}{M}$.

The complexity of Pinwheel Scheduling has been open since it was introduced. It was only recently shown by Jacobs and Longo [25] that there is no pseudopolynomial time algorithm solving the problem unless SAT has an exact algorithm running in expected time $n^{O(\log n \log\log n)}$, implying for example that the randomized exponential time hypothesis fails to hold [8, 15]. Since the latter is unlikely, one could conclude that Pinwheel Scheduling is not solvable in pseudopolynomial time. It remains open whether the problem is PSPACE-complete.

Similar to Pinwheel Scheduling, the k-server Periodic Maintenance Problem [3, 17, 27] has n jobs. Each of the k servers may serve at most one job per time unit/day. Each job j has a specified periodicity m_j and a processing time c_j. However, job j is required to be served exactly every m_j days apart rather than within every m_j days. The case $k = 1, c_j = 1$ for all j is analogous to Pinwheel Scheduling, except for the exact periodicity constraint. For any $k \geq 1$, Mok et al. [27] have shown it is NP-complete in the strong sense. For the special case when m_j are multiples of each other or when there are at most 2 different periodicities, they have shown it is in P.

Other related problems with a compact input representation include real-time scheduling of a set of recurrent *sporadic* tasks [2, 5], where two consecutive jobs of the same task are released at least some time interval apart and each of them has a processing time and a deadline.

On a single machine, EDF (Earliest Deadline First) answers correctly if the task system is feasible or not. However, because of the concise input, the complexity of this question remained open for a long time. Only recently it has been proved that deciding feasibility on a single processor is coNP-hard, even if the utilization is bounded [18].

Another related problem is the (discrete) BAMBOO GARDEN TRIMMING PROBLEM introduced by Gasieniec et al. [23]. There are n bamboos, each having a given growth rate, which may be viewed as inducing a periodicity. On each day, a robot may trim at most one bamboo back to height 0. The goal is to minimize the maximum height of the bamboos. Gasieniec et al. provide a 2-approximation. The approximation ratio has subsequently been improved to 1.888 [16] and $\frac{12}{7}$ [28]. The problem was also considered by Bilò et al. [4], who studied the approximation ratio of simpler algorithms, like cutting down the largest bamboo every day.

This paper. We investigate the computational complexity of both MIN–MAX RFTT and MIN–AVG RFTT. Apart from an NP-hardness proof for MIN–MAX RFTT on a tree, most of the complexity is related to the complexity of PINWHEEL SCHEDULING. Some interesting inapproximability results follow from this relation. After that, we will start with some special cases. In Sect. 3, we give our most remarkable result, a constant factor approximation for MIN–MAX RFTT on a tree, next to a less remarkable constant factor approximation for the version of MIN–AVG RFTT on a tree. In the same section, we show for MIN–AVG RFTT, that the problem can be solved to optimality in pseudopolynomial time on line metrics. Finally, in Sect. 4, we present logarithmic factor approximations for both problem versions on general metrics.

2 Complexity

In this section, we investigate the computational complexity for both objectives. First note that in case all turnover times are equal to 1, both MIN–AVG and MIN–MAX RFTT are equivalent to the TRAVELING SALESMAN PROBLEM (TSP).

Moreover, as Theorem 1 shows, the MIN–MAX RFTT remains NP-hard even on star graphs (where TSP is trivial) using a reduction from 3-PARTITION. Here, we are given $3m$ integers $a_1, \dots, a_{3m}$ such that $\frac{B}{4} < a_i < \frac{B}{2}$ for all i, where $B = \frac{1}{m}\sum_i a_i$. The question is whether we can partition the integers into m sets of three integers that add up to B [22].

Theorem 1 MIN–MAX RFTT *is NP-hard on star graphs.*

Proof Given an instance of 3-PARTITION, create a weighted star graph G with the root at the center and for every integer a_i a leaf vertex attached to the center with an edge with cost $a_i/2$. Finally set the turnover time to m for every leaf. We will show that the RFTT instance has value B if and only if we have a YES-instance for 3-PARTITION.

Given a valid 3-partition of the integers, clearly one can assign every set of 3 integers a unique day in $1, \dots, m$ and visit the associated leaves on every multiple of that day for a valid RFTT solution. For the opposite direction note that $\frac{B}{4} < a_i$ for all i. Hence, we cannot visit more than 3 vertices on one day. Since after m days all vertices must have been visited due to the turnover times, it follows that the first m days of the schedule correspond to a valid 3-partition. □

Interestingly, we can also relate the complexity of MIN–AVG and MIN–MAX RFTT to the complexity of PINWHEEL SCHEDULING. We note that PINWHEEL SCHEDULING is neither known to be NP-hard nor in NP, although it is conjectured to be PSPACE-complete. We also note that reductions to PINWHEEL SCHEDULING are used in [1] to classify the complexity of some capacitated inventory routing problems. Next, we show that for $\alpha < 2$, there is no α-approximation for MIN–MAX RFTT on star graphs, assuming that PINWHEEL SCHEDULING is intractable.

Theorem 2 *For $\alpha < 2$, there is no α-approximation for* MIN–MAX RFTT *on star graphs, unless* PINWHEEL SCHEDULING *is polynomially solvable.*

Proof Given an instance of PINWHEEL SCHEDULING, construct a star graph with the root at the center, each leaf corresponding to a job in the PINWHEEL SCHEDULING-instance. The turnover time of the leaves is equal to the periods of the corresponding jobs in the PINWHEEL SCHEDULING-instance. Furthermore, all edges have cost equal to 1. If the instance of PINWHEEL SCHEDULING is a yes-instance, then we have a feasible schedule that visits one leaf per day. Hence, the tour cost is equal to 2 each day. If the instance of PINWHEEL SCHEDULING is a no-instance, we have to visit at least two leaves on some day in order to get a feasible schedule. Hence, the maximum tour cost is at least 4. Thus, an α-approximation, with $\alpha < 2$, for MIN–MAX RFTT could distinguish between yes- and no-instances of PINWHEEL SCHEDULING. □

Using a reduction from PINWHEEL SCHEDULING, we can also show that MIN–AVG RFTT is hard on series-parallel graphs.

Theorem 3 PINWHEEL SCHEDULING *polynomially reduces to* MIN–AVG RFTT *on series-parallel graphs.*

Proof Given an instance $p_1, ..., p_n$ of PINWHEEL SCHEDULING, create an instance (G, τ) of MIN–AVG RFTT. We define

$$V = \{s, w^1, w, w^2\} \cup V^1 \cup V^2,$$

where $V^i = \{v_1^i, ..., v_n^i\}$ for $i = 1, 2$,

$$E = \{(s, v) | v \in V^1 \cup V^2\} \cup \{(w^i, v_j^i) | j = 1, \dots, n; i = 1, 2\} \cup \{(w^1, w), (w^2, w)\},$$

and $\tau_{w^1} = \tau_{w^2} = \tau_w = 1$, $\tau_{v_j^1} = \tau_{v_j^2} = p_j$. All edge costs are 1 and s is the root. See Fig. 1 for an illustration.

We claim that the instance (G, τ) has a solution of average cost at most 6 if and only if $p_1, ..., p_n$ is a feasible PINWHEEL SCHEDULING instance.

For the 'if'-direction, suppose we have a feasible pinwheel schedule. Then we create a replenishment schedule as follows: we take as the set of vertices visited on day k, $J_k = \{v_j^1, w^1, w, w^2, v_j^2\}$, where job j is the job scheduled on day k in the pinwheel solution (when no job is visited, pick an arbitrary vertex in V^1 and V^2). The pinwheel schedule then guarantees that the periods of vertices in $V^1 \cup V^2$ are satisfied, while vertices w^1, w, w^2 are visited every day, as required. Now since any tour $(s, v_j^1, w^1, w, w^2, v_j^2, s)$ has cost 6, we can do this within the claimed cost.

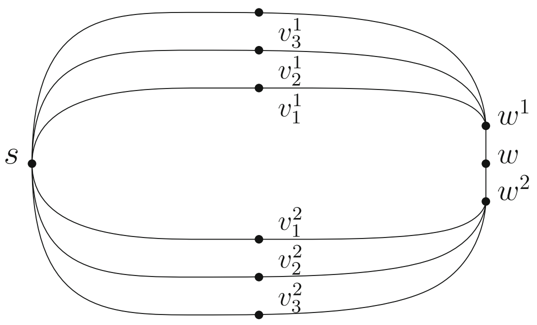

Fig. 1 Instance created in the proof of Theorem 3

For the 'only if'-direction, note that any replenishment schedule must have cost 6 at least, since no tour that visits w^1, w, and w^2 costs less than that. Moreover, any tour visiting those three vertices that is not of the form $(s, v_j^1, w^1, w, w^2, v_j^2, s)$, will cost strictly more than 6. It follows that if the average cost of the replenishment schedule is 6, every tour visits exactly one vertex from both V^1 and V^2. Since $\tau_{v_j^1} = p_j$ for all j, this directly implies that the PINWHEEL SCHEDULING instance is feasible. □

3 Approximation on Trees

In this section we give a 2-approximation for MIN–AVG RFTT and a 6-approximation for MIN–MAX RFTT on trees. We start out by showing we lose a factor of 2 in the approximation guarantee if we round each turnover time down to a power of 2.

Lemma 1 *Given an instance (G, τ) of RFTT, let τ' be obtained by rounding every turnover time in τ down to a power of 2. Then $OPT(G, \tau') \le 2OPT(G, \tau)$ for both MIN–AVG and MIN–MAX objectives.*

Proof Let $(G, \bar{\tau})$ denote the instance found from (G, τ) by rounding every turnover time up to a power of 2. Since any schedule remains feasible if we *round up* the turnover times, we have that $OPT(G, \bar{\tau}) \le OPT(G, \tau) \le OPT(G, \tau')$ for both objectives.

Suppose we have an optimal solution for $(G, \bar{\tau})$ for any of the MIN–AVG or the MIN–MAX version. Let $\bar{T}_k$ denote the tour at day k. We can construct a feasible schedule for (G, τ') by scheduling the concatenation of $\bar{T}_{2k-1}$ and $\bar{T}_{2k}$ on day k. In case of the MIN–MAX version, the maximum tour cost in this schedule is at most twice that of the optimal solution for $(G, \bar{\tau})$, i.e., at most $2OPT(G, \bar{\tau})$, which is at most $2OPT(G, \tau)$. For the MIN–AVG version notice that all tours of the original schedule are executed in half the number of days. Hence, the new schedule has average cost at most $2OPT(G, \bar{\tau}) \le 2OPT(G, \tau)$. □

In the remainder we assume w.l.o.g. that every vertex has a turnover time, and that G is rooted at s. Since any tour visiting a vertex automatically visits all the vertices on the unique path from the root to this vertex, we assume w.l.o.g. that turnover times are non-decreasing on any path from the root to a leaf vertex in G. Furthermore, for an edge e in E we define $D(e)$ to be the set of vertices that are a descendent of e. We also need the following definition.

Definition 1 (*tt-weight of an edge*) For any edge e in G we define:

$$q(e) = \min_{v_j \in D(e)} \tau_j.$$

We call this quantity the tt-weight (turnover time-weight) of e.

So, under the assumption we make on G, $q(e)$ is equal to the turnover time of the incident vertex of e furthest from the root. This definition allows us to express the following lower bound, recalling that $c(e)$ is the cost of edge e.

Lemma 2 (tt-weighted tree) *For an instance* (G, τ) *of the* RFTT *on trees it holds that the average tour cost is at least:*

$$L(G, \tau) := 2 \sum_{e \in E} \frac{c(e)}{q(e)}.$$

Proof Note that edge e has to be traversed, back and forth, at least once every $q(e)$ days in order to satisfy the turnover times of its descendants. Hence, it contributes at least $2c(e)/q(e)$ to the average tour cost. Summing over all edges gives the desired lower bound. □

Since the maximum tour cost is at least the average tour cost, Lemma 2 also provides a lower bound for the MIN–MAX objective.

An approximation for MIN–AVG RFTT is found by rounding down all turnover times to powers of 2 and then visit each vertex j on every day that is a multiple of the rounded turnover time τ'_j.

Theorem 4 *There is a 2-approximation for* MIN–AVG RFTT *on trees.*

Proof Consider an edge e. Since all turnover times in the rounded instance are powers of 2, the turnover times of the descendents of e are multiples of $q(e)$. By visiting each vertex j on every day that is a multiple of τ'_j, edge e is traversed exactly once in every $q(e)$ consecutive days. Since this holds for any edge, the average tour cost of the solution is equal to $L(G, \tau')$. Hence, by Lemma 2 this solution is optimal for this rounded instance. Since Lemma 1 says that this rounding procedure loses us a factor of 2 in the approximation, we have obtained a 2-approximation for MIN–AVG RFTT on trees. □

3.1 MIN-MAX

We will now show that we can achieve a 6-approximation for MIN–MAX RFTT on trees. The main ingredient is a 3-approximation algorithm for the case that all turnover times are powers of 2, the general case then follows by applying Lemma 1.

We first present an introductory high-level description explaining the overall structure and idea of the algorithm. The algorithm produces a solution by recursively fixing the schedules of vertices with increasing turnover times; namely the algorithm computes for each vertex v values $h(v)$ and $p(v)$ that denotes the day of the first visit and the number of days between two consecutive visits, respectively. At the top level, it fixes all vertices v_j with $\tau_j = 1$, who need to be visited daily from which we will have $h(v) = p(v) = 1$. It then splits the remaining vertices into two new instances, each containing only vertices with turnover time at least 2. One instance will use only odd days and the other one only even days.

We want the recursive splitting of the subtrees to distribute the vertices evenly over the days, in the sense that each day must get a subtree of approximately equal cost. To guide this we use an optimal TSP tour in the tree that we compute prior to the start of the algorithm. This tour is split into two parts in each recursive step, and each part is passed on to one of the child calls of the algorithm. The set of vertices covered by the part of the tour a child receives determines which vertices are in its instance. Since every child receives a connected part of the tour, the subtree induced by the covered set of vertices will form a single connected component, implying it is sensible to visit vertices there simultaneously.

The most important aspect of making this work is to ensure that the tour is split in a balanced way. In particular, we want its cost to be balanced, taking into account the tt-weights. Before we describe more precisely what this means, we will introduce some formal definitions.

Preliminaries In the context of this section, a walk $W = (v_1, e_1, v_2, \dots, v_k)$ is given by an alternating sequence of vertices and edges, each edge between its two incident vertices. A TSP tour is a walk that contains every vertex in G and starts and ends with the root. We use $v \in W$ or $e \in W$ to denote that a vertex or edge occurs (at least once) in the walk.

Definition 2 Let $W = (v_1, \dots, v_j, e_j, v_{j+1}, \dots, v_k)$ be a walk in the tree G. A split of W is a pair of walks $W_{pre} = (v_1, e_1, v_2, \dots, v_j)$ and $W_{suf} = (v_{j+1}, e_{j+1}, v_{j+2}, e_{j+2}, \dots, v_k)$. We call these walks the prefix and the suffix, respectively.

Definition 3 A split is *tt-balanced* if:

$$\sum_{e \in W_{pre}} c(e)/q(e) \le \frac{1}{2} \sum_{e \in W} c(e)/q(e) \quad \text{and} \quad \sum_{e \in W_{suf}} c(e)/q(e) \le \frac{1}{2} \sum_{e \in W} c(e)/q(e).$$

Since the tt-weights are a measure of how often an edge is visited, a balanced split essentially partitions the work equally between the two parts.

We also need the concept of edge contraction, which is defined in Definition 4, and illustrated in Fig. 2.

Definition 4 Let $e \in G$ be an edge, with endpoints u, v. The operation of contracting e in G, merges u and v into a single vertex, that is adjacent to every vertex that was adjacent to u or v.

We denote the contracted graph by G/e. If F is an edge set in G, G/F denotes the graph found by contracting all edges in F (in any order). Similarly if W is a walk in G, we denote W/F the walk in the contracted graph G/F.

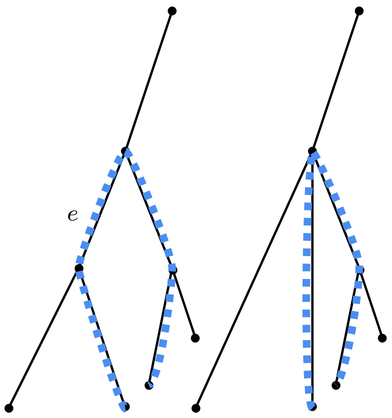

Fig. 2 Illustration of contracting edge e. Note that we only allow contraction of adjacent vertices. This means that for every walk W and edge e in G, there is an obvious mapping to a walk in G/e, as shown by the blue dotted path here

Decontraction is simply the operation that reverses a previously executed edge contraction.

In a call of procedure MaxTreeSchedule the algorithm receives, among the input parameters, a walk W and a list of unscheduled vertices U from its parent call, and integers t and k. In the first call it receives a TSP tour and the values $t = 1$ and $k = 0$. If the walk W contains at least one edge then the call considers the unscheduled vertices in the walk, schedules the first visit to every vertex v_j for which $\tau_j = 2^k$ on day t, sets the time between two consecutive visits to vertex v_j equal to 2^k, and then removes these vertices from the list U. Next, it contracts edges in G with $q(e) = 2^k$, and walk W is adjusted to a walk in the contracted metric by deleting the contracted edges from the walk. It now computes a tt-balanced split W_0, W_1 of W', where W' is the walk in the contracted metric, and passes each one to a separate child call of the algorithm. Each child call returns a schedule that visits vertices with period at least 2^{k+1}. This will lead to a schedule such that all vertices visited during the first call ($k = 0$), if any, are visited daily and all vertices scheduled by the first (second) child of the first call get scheduled on odd (even) days.

Algorithm 1 provides a description of the main algorithm, whereas Algorithm 2 provides a detailed description of the subroutine. The main algorithm takes as input an instance (G, τ) with τ powers of two, and computes for each vertex v values $h(v)$ and $p(v)$, that represent the day of the first visit of v and the number of days between two consecutive visits, respectively. Since our metric is a tree, the optimal solution for each day is a depth-first search. Hence, this information is sufficient to represent the route for each day.

Algorithm 1 Approximation algorithm for MIN–MAX RFTT on trees

```
1: Compute an optimal TSP tour $W$ in $G$, the tt-weights $q$ and set $U \leftarrow V$.
2: MaxTreeSchedule($G, U, W, 1, 0$).
3: Return for each vertex $v$ values $h(v)$ and $p(v)$ that store the day of the first visit and the period between
   two consecutive visits.
```

Algorithm 2 MAXTREESCHEDULE(G, U, W, t, k)

```
1: if $W$ contains a single vertex $v$ then return
       $$h(v) = t,\ p(v) = \tau_v.$$
2: if $W$ contains two or more vertices then determine the set of vertices $X$ with turnover time $2^k$ and edges
   connecting them (note that $X$ can be the empty set) then
       $$h(v) = t,\ p(v) = 2^k \text{ for all } v \in X$$
       $$X \leftarrow \{v_j \in U : v_j \in W \text{ and } \tau_j = 2^k\} \quad \text{and} \quad Y \leftarrow \{e \in W : q(e) = 2^k\}.$$
3: Remove the covered vertices from $U$ and contract edges $Y$
       $$U' \leftarrow U \setminus X \quad \text{and} \quad G' \leftarrow G/Y.$$
4: Create $W_0$, $W_1$ as $tt$-balanced split of $W/Y$.
5: Compute the child schedules recursively
       MAXTREESCHEDULE$(G', U', W_0, t, k+1)$, MAXTREESCHEDULE$(G', U', W_1, t+2^k, k+1)$
```

Feasibility First of all we observe that, at the end of the algorithm, values $h(v)$ and $p(v)$ are computed for each vertex. Feasibility of the schedule follows by observing that every vertex with turnover time 2^k is scheduled for the first time at time t, $t \leq 2^k$, and then periodically every $\tau_k = 2^k$ days.

Cost analysis In the sequel we will prove that the approximation ratio between the cost of the solution returned by Algorithm 1 and that of the optimum is constant. We first note that step 1 is essential for us to bound the running time of the algorithm when the procedure is called in case W does not contain an edge, but that this step is not essential for the cost analysis. We observe that if we modify the recursive procedure by eliminating step 1 and introducing a stopping rule when W is empty, and therefore recursively applying the procedure in steps 2 to 5 when W is a singleton vertex, then we obtain the same solution. Hence, in the cost analysis, we can assume that every call to MAXTREESCHEDULE uses the procedure in steps 2 to 5.

The height of a tree Height(G) is the maximum cost of a root-leaf path in G. Note that every leaf of the tree represents a vertex that must be visited, and hence 2Height(G) is a lower bound on OPT, the optimal value. We will prove that the cost of a tree connecting the vertices on any day t is bounded from above by:

$$L(G, \tau) + \text{Height}(G).$$

After doubling the edges of the tree to get a tour this gives a cost of $2L(G, \tau) +$ 2Height(G). By Lemma 2, this is at most $3OPT$, as required.

It will be more convenient to look at leaves of the recursion instead of days though. Define a leaf-call as a call to MAXTREESCHEDULE that does not have any child calls, i.e., a call for which $U' = \emptyset$. Note that the set of vertices scheduled at day t by the procedure is generated from the set X scheduled in a leaf-call of the recursion together with all the sets scheduled by ancestor calls of that leaf-call. Hence, if we can bound the cost of every set of vertices generated by a leaf-call and its ancestors, we immediately bound the cost of the sets scheduled on every day.

In the rest of this section, we will use the following conventions. G^{un} will always refer to the original (uncontracted) metric. Given a call to MaxTreeSchedule, X and Y refer to the sets defined in Step 2 of that call.

We say that a set of edges E' spans a set of vertices V' if the edges in E' form a spanning tree containing all vertices in V'. We will start with the following (visually intuitive) observation illustrated in Fig. 3.

Lemma 3 *In any call to* MaxTreeSchedule*, Y spans X (in the contracted tree).*

Proof Assume $|X| > 1$ otherwise there is nothing to be done. Any shortest path between vertices in X must be completely made up of edges e with $q(e) = 2^k$, and therefore is completely contained in $Y = \{e \in W : q(e) = 2^k\}$. So, Y spans X. □

In any call to MaxTreeSchedule, let Y^{anc} be the set of edges that have been contracted so far by ancestor calls. Then, we have the following lemma.

Lemma 4 *The set $Y^{anc} \cup Y$ spans a set of vertices that contains X in G^{un}.*

Proof By Lemma 3, Y spans X in the contracted tree G. Now, if we uncontract any single previously contracted edge e, then $Y \cup \{e\}$ spans a set of vertices containing X in the uncontracted graph. The lemma follows by repeatedly applying this principle to every edge in Y^{anc}. □

So, we only need to connect the connected components in $Y^{anc} \cup Y$, at most one for each ancestor call, to the root. The next lemma will help in showing that we can do this at the cost of a single root-leaf path.

Lemma 5 *In any call to MaxTreeSchedule, if a non-empty set of edges Y is contracted, the vertex $v \in W$ that is closest to the root is also in the subtree spanned by Y, and has the property that all vertices $v_\ell \in W$ with $\tau_\ell > 2^k$ are descendants of v in G.*

Proof Notice that either, W contains the vertex of G that is obtained after contracting Y^{anc}, which is in fact the root of G and is picked as v, or, if that vertex is not in W then $v \in W$ is the root of the subtree that is traversed in a depth-search manner by W. Since Y is contracted it must contain vertices with turnover times 2^k. Hence, given our assumption that turnover times of vertices are non-decreasing on any path directed away from the root, any vertex $v_\ell \in W$ with $\tau_\ell > 2^k$ must be a descendent of v in G. □

Lemma 6 *In any call to MaxTreeSchedule that contracts a non-empty set of edges Y, let v_ℓ be an arbitrary vertex in X and P a shortest path from v_ℓ to the root in G^{un}. Let Y^{anc} be the set of edges contracted and X^{anc} the set of vertices scheduled in ancestor calls.*

Then $Y^{anc} \cup Y \cup P$ contains a tree connecting $X \cup X^{anc}$ to the root of G^{un}.

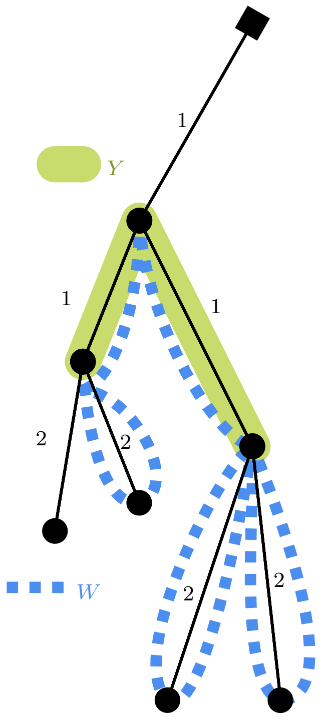

Fig. 3 Illustration of Lemma 3. The number next to each edge e represents its tt-weight $q(e)$. The blue dotted indicates the walk W received by the call. The emphasized fat green edges are in Y and will be contracted in the current iteration

Proof By Lemma 5, any previously contracted set of edges is incident to a vertex v_j that is an ancestor of v_ℓ, and therefore lies on P. It follows that P connects every connected component in $Y^{anc} \cup Y$ to the root. Using Lemma 4 it now follows that $Y^{anc} \cup Y \cup P$ contains a tree spanning every vertex scheduled by an ancestor call plus the root. □

It remains to show that the cost of $Y^{anc} \cup Y \cup P$ is at most $L(G^{un}, \tau) + \text{Height}(G^{un})$. For this, we need the following lemma.

Lemma 7 *In any call to* MaxTreeSchedule, *let Y^{anc} be the set of edges that have been contracted so far by ancestor calls. Then*

$$\sum_{e \in Y^{anc}} c(e) + 2^k \sum_{e \in W} c(e)/q(e) \leq L(G^{un}, \tau).$$

Proof We prove this by induction on the depth of the recursion. For the base case $(k = 0)$ it clearly holds since Y^{anc} is empty and W is a TSP tour, where, as it contains every edge twice, we have

$$\sum_{e \in W} c(e)/(q(e) = 2 \sum_{e \in E} c(e)/(q(e) = L(G^{un}, \tau).$$

Suppose the statement of the lemma holds for depth k. We will prove it holds for depth $k+1$, i.e., for $i = 0, 1$:

$$\sum_{e \in Y^{anc} \cup Y} c(e) + 2^{k+1} \sum_{e \in W_i} c(e)/q(e) \leq L(G^{un}, \tau).$$

Since $q(e) = 2^k$ for all $e \in Y$, and since W_0, W_1 is a tt-balanced split of W/Y, we have for $i = 0, 1$:

$$\begin{aligned}\sum_{e \in Y} c(e)/2^k &= \sum_{e \in Y} c(e)/q(e) \\ &= \sum_{e \in W} c(e)/q(e) - \sum_{e \in W/Y} c(e)/q(e) \\ &\leq \sum_{e \in W} c(e)/q(e) - \sum_{e \in W_i} 2c(e)/q(e).\end{aligned}$$

Hence, we get for $i = 0, 1$:

$$\begin{aligned}&\sum_{e \in Y^{anc} \cup Y} c(e) + 2^{k+1} \sum_{e \in W_i} c(e)/q(e) \\ &= \sum_{e \in Y^{anc}} c(e) + \sum_{e \in Y} c(e) + 2^{k+1} \sum_{e \in W_i} c(e)/q(e) \\ &\leq \sum_{e \in Y^{anc}} c(e) + \sum_{e \in Y} c(e) + 2^k \sum_{e \in W} c(e)/q(e) - \sum_{e \in Y} c(e) \\ &\leq \; L(G^{un}, \tau)\end{aligned}$$

□

This brings us to the main result of our cost bound.

Lemma 8 *Let S be the schedule returned by Algorithm 1 on instance (G, τ). For any day t, the maximum cost of a shortest tour visiting all vertices to be visited at day t, is at most*

$$2L(G, \tau) + 2Height(G).$$

Proof Since the cost of any root-leaf path is at most Height(G^{un}), it is enough to show that the cost of the contracted edges $Y^{anc} \cup Y$ is at most $L(G^{un}, \tau)$, i.e.,

$$\sum_{e \in Y^{anc} \cup Y} c(e) \leq L(G, \tau).$$

Since $q(e) = 2^k$ for all $e \in Y$, in any call to MaxTreeSchedule we have:

$$\sum_{e \in Y} c(e) = 2^k \sum_{e \in Y} c(e)/q(e) \leq 2^k \sum_{e \in W} c(e)/q(e).$$

Using this fact and Lemma 7, we have:

$$\begin{aligned}\sum_{e\in Y^{anc}\cup Y} c(e) &= \sum_{e\in Y^{anc}} c(e) + \sum_{e\in Y} c(e)\\ &\le \sum_{e\in Y^{anc}} c(e) + 2^k \sum_{e\in W} c(e)/q(e)\\ &\le L(G^{un}, \tau).\end{aligned}$$

Since a tour passes through all edges twice, the lemma follows. □

Observe that every split of a walk removes one edge and if there are no edges in W then no further calls are made. Therefore the number of recursive calls is bounded by $2|E|$ where $|E|$ denotes the number of edges.

Hence, by Lemma 8 we obtained a 3-approximation for MIN–MAX RFTT on trees if turnover times are powers of 2. To apply Algorithm 1 to general instances, we need to round the turnover times, losing a factor of two on the bound according to Lemma 1. Taking all this into account, we state our main result.

Theorem 5 *There is a* 6*-approximation for* MIN–MAX RFTT *on trees.*

3.2 MIN-AVG RFTT on the Line

As an even more special underlying metric, we might consider MIN–AVG RFTT on the line (on a path). For the MIN–MAX version this case is trivial, but for the MIN–AVG version its complexity is unclear: we do not know whether it is in NP, although we expect it to be NP-hard. On the positive side we can show that the problem is not strongly NP-hard.

In this section we will show how to solve MIN–AVG RFTT on the line in pseudopolynomial time. Since we are minimizing the average, we can reduce this problem to two times the MIN–AVG RFTT on the half-line (a path with the root in one of the leaves). To see this, note that any solution on the line is a concatenation of two solutions, one for each side of the line. On the positive half-line each vertex v_i has a distance $d_i \in \mathbb{N}$ from the origin. Suppose vertices are numbered such that $d_1 \le \ldots \le d_n$. We present a pseudopolynomial time dynamic programming algorithm for this problem, based on the following observations.

First of all, we note that any tour visiting vertex v_j automatically visits every vertex v_i with $i < j$. As in the tree case, we therefore assume that $\tau_i \le \tau_j$ for $i < j$. Thus, after visiting v_j, all v_i with $i \le j$ have a remaining turnover time of τ_i. In particular, after visiting vertex v_n, each vertex v_i has a remaining turnover time equal to τ_i. Hence, if we visit vertex v_n for the first time on day L, we can create a schedule by repeating the obtained schedule up to day L. For the dynamic program to work, we guess L, and try all guesses between 1 and τ_n.

The dynamic program now works as follows. Suppose we are given the optimal solution for vertices $v_1, \ldots, v_{i-1}$ when only considering the days $1, \ldots, k$. Now we want to include v_i in the optimal solution for the first k days. If $k < \hat{\tau}_i := \min\{\tau_i, L\}$, it is not necessary to visit v_i during the first k days, and hence it is optimal to take the optimal solution for the first $i-1$ vertices and k days. Otherwise, we need to visit v_i on some day ℓ in $\{1, \ldots, \hat{\tau}_i\}$. Before day ℓ, we only need to visit the vertices $v_1, \ldots, v_{i-1}$. Thus, we take the optimal $\ell-1$ tours for visiting the first $i-1$ vertices in the first $\ell-1$ days. After day ℓ, all vertices have the same remaining turnover time as they had at time zero. Hence, we can take the optimal tours for the first i vertices and $k-\ell$ days.

Let $\phi_L(i,k) :=$ the minimum cost of the first k tours visiting vertices v_1, ..., v_i. We initialize $\phi_L(0,k) = \phi_L(i,0) = 0$ and we use the recursion:

$$\phi_L(i,k) = \begin{cases} \phi_L(i-1,k) & \text{if } k < \hat{\tau}_i, \\ \min_{\ell=1,\ldots,\hat{\tau}_i} \phi_L(i-1,\ell-1) + d_i + \phi_L(i,k-\ell) & \text{otherwise.} \end{cases}$$

For a given L, the average cost of the optimal solution, given that vertex n is visited on day L, is equal to $\phi_L(n,L)/L$. Hence, the optimal solution is the schedule that corresponds to the value $L \in \{1, \ldots, \tau_n\}$ minimizing $\phi_L(n,L)/L$. Note that the algorithm runs in time $O(n\tau_n^3)$, implying the following.

Theorem 6 MIN–AVG RFTT *on the line can be solved in pseudopolynomial time.*

4 Approximation on General Graphs

We will now present logarithmic approximations for both objectives. To facilitate the exposition, we omit the description of how to obtain $h(v)$ and $p(v)$ in each of the algorithms. In what follows, we use log to denote the logarithm with base 2.

Theorem 7 *There exists an* $O(\log \tau_{max})$*-approximation for* MIN–MAX *and* MIN–AVG RFTT.

Proof By Lemma 1 we may assume every τ_i is a power of 2 so that there are at most $\log \tau_{max}$ different turnover times. We simply treat the sets of vertices with the same turnover time as separate instances and concatenate the solutions. Our result then follows from the fact that for each of these instances a constant factor approximation is available. In the case of the MIN–MAX objective we get the k-TSP problem [21], where k is equal to the turnover time of the vertices in the instance. In the case of MIN–AVG, we need to minimize the sum over all k tours. But since all turnover times are equal there is no advantage to visiting vertices on different days, hence we recover a simple TSP problem on these vertices. □

In the case of the MIN–MAX objective, it is relatively simple to adapt this idea to obtain an $O(\log n)$-approximation by appropriately reassigning vertices to lower turnover times, as per Theorem 8.

Theorem 8 MIN–MAX RFTT *has an* $O(\log n)$*-approximation.*

Proof We start by assuming that every turnover time is a power of 2. Next, we split up the instance into two new instances. To this end we first define a turnover time k to be *saturated* if $|\{v_j \in V | \tau_j = k\}| \geq k$. In the first instance we retain the set of vertices V_1 with saturated turnover times, and in the second all vertices V_2 with unsaturated turnover times. Now if all turnover times are saturated, as is the case in V_1, then $\tau_{max} = O(n)$ and we can find an $O(\log n)$-approximation using Theorem 7. So what remains is to find an $O(\log n)$-approximation for the second instance.

Since in V_2 no turnover time is saturated, it is easy to see that we can partition the vertices in V_2 into $\lceil \log n \rceil$ sets $X_1, X_2, X_4, \ldots, X_{2^{\lceil \log n \rceil}}$, such that $|X_i| \leq i$, and such that $\tau_j \geq i$ for all $v_j \in X_i$. For example we could first add all vertices v_j with $\tau_j = i$ to X_i for each $i \leq 2^{\lceil \log n \rceil}$, and then arbitrarily distribute vertices v_j with $\tau_j > 2^{\lceil \log n \rceil}$ among the sets that have space. Note that assigning a vertex to X_i implies that the turnover time is rounded to i. We now produce a schedule by visiting all vertices in any set X_k on different days. Since turnover times are rounded down, and since there is enough space for n vertices in $X_1, X_2, X_4, \ldots, X_{2^{\lceil \log n \rceil}}$, this is feasible. Moreover, at most $\lceil \log n \rceil$ vertices are visited on a given day, which leads to an $O(\log n)$-approximation factor, as required. □

The approach of Theorem 7 does not trivially extend to the MIN–AVG case. However, applying the FRT tree embedding [19] of the initial instance and then using the 2-approximation for tree metrics to obtain the final solution leads straightforwardly to a randomized $O(\log n)$-approximation.

A more direct, and deterministic $O(\log n)$-approximation is possible as well. In particular, we use the simple heuristic of visiting each vertex on every day that is a multiple of its turnover time, when turnover times are powers of 2. We call such a schedule a *synchronized* solution, and show that this gives a logarithmic approximation. We will describe the details of the algorithm while proving the following theorem.

Theorem 9 MIN–AVG RFTT *has a deterministic $O(\log n)$-approximation.*

We start the algorithm by rounding all periods to powers of 2 and delay visiting any vertex for as long as possible. Next, for every day we use any constant factor approximation for TSP to calculate a tour on the vertices whose visit can no longer be delayed. We call this a *synchronized* solution. It takes some work to show this does indeed provide a logarithmic approximation. We do this by showing that any synchronized solution is no more costly than a *non-decreasing* solution, in which every tour is based on a tree that has the vertices ordered by ascending turnover times from root to leaves. We then show that such a non-decreasing solution costs at most $O(\log n)$ times the optimal solution, and also provide an example showing this ratio can be as large as $\Omega(\log \log n)$. Moreover, we show that the optimal non-decreasing solution is at most twice as costly as the optimal synchronized solution. This also implies that that any better (e.g. constant factor) approximation algorithm must avoid finding such solutions.

As always we will assume that turnover times are rounded down to powers of 2. Let us define a *synchronized* solution, as one where each vertex with turnover time 2^i is visited on each day that is a multiple of 2^i, for all i. We define a *non-decreasing* tree as a tree on the root and a subset of vertices, such that the turnover times on every path from the root to the leaves are non-decreasing. A non-decreasing solution is a solution in which for each day the tour is given by visiting the vertices of a non-decreasing tree in depth first order.

The following two lemmas show that optimal synchronized and non-de-creasing solutions differ in cost by at most a constant factor.

Lemma 9 *For any non-decreasing solution there exists a synchronized solution with at most twice the cost.*

Proof Suppose we have a non-decreasing solution. Let T_i be the non-decreasing tree associated with day i, for $i \in \mathbb{N}$. We will show that we can find a tree T_i' for each $i \in \mathbb{N}$ that costs at most as much as T_i on average, and such that any vertex v appears in tree T_i' if i is a multiple of τ_v. A synchronized solution can then be found by taking the tour on day i to be a depth first search trail in T_i', thus losing a factor 2 in the cost of the trail w.r.t. the tree.

Iteratively, from $j = 0, \dots, \log \tau_{max}$, we build the trees for the synchronized solution. In iteration j and for all i, select all unmarked edges in T_i that are used on some path from the root to a vertex v with $\tau_v = 2^j$, and mark them. Then insert these edges in the tree T_k' for the earliest following day k that is a multiple of 2^j, so $k = \lceil \frac{i}{2^j} \rceil 2^j$.

We now show by induction that after iteration j, for each day k that is a multiple of 2^j, T_k' is a tree connecting the root to all vertices v with turnover time 2^j. The base case $j = 0$ follows from the fact that the trees T_i are non-decreasing and must contain every vertex with turnover time 1. For higher j, it is easy to see that T_k' must contain a path from v to w for all vertices v with $\tau_v = 2^j$ and some w with $\tau_w < 2^j$. But T_k' already contains w by our inductive hypothesis and the result follows. □

We have the corresponding converse result, as per Lemma 10.

Lemma 10 *The optimal non-decreasing solution costs at most two times the optimal synchronized cost.*

Proof Recall that we rounded down all turnover times to powers of 2, and that we defined a synchronized solution as one where each vertex with turnover time 2^i is visited on each day that is a multiple of 2^i. Hence, we may assume that any synchronized solution uses at most $\log \tau_{max} + 1$ distinct tours, let us label them $T_0, \dots, T_{\log \tau_{max}}$, where T_j visits all vertices with turnover time at most 2^j. Furthermore define $\Delta_0 = c(T_0)$ and $\Delta_j = c(T_j) - c(T_{j-1})$, for $j = 1, \dots, \log \tau_{max}$. Then it holds that the cost of the synchronized solution is

$$c_{sync} = \sum_{j=0}^{\log \tau_{max}} \frac{1}{2^j} \Delta_j.$$

To see this, consider the following. We have to visit vertices with turnover time 1 each day. To account for this, we add $c(T_0) = \Delta_0$ to the objective value. Once every two days, we have to visit vertices with turnover time 2, which are visited by T_1 together with the vertices with turnover time 1. So, on these days we do not have to use T_0. Hence, we add $\frac{1}{2}(c(T_1) - c(T_0))$ to the objective value. Repeating this reasoning gives the desired formula.

Now we can create a non-decreasing solution for any day with 2^j as its largest power of 2 from the synchronized solution, by taking the concatenation of $T_0, \dots, T_j$ while short-cutting already visited vertices. In other words, we first follow tour T_0, then tour T_1 without the vertices with turnover time 1, then tour T_2 without vertices with turnover time 1 and 2, etc. Now the cost of such a solution is

$$\begin{aligned} c_{ndecr} &\leq \sum_{j=0}^{\log \tau_{max}} \frac{1}{2^j} c(T_j) \\ &= \sum_{j=0}^{\log \tau_{max}} \frac{1}{2^j} \sum_{i=0}^{j} \Delta_i \\ &= \sum_{i=0}^{\log \tau_{max}} \sum_{j=i}^{\log \tau_{max}} \frac{1}{2^j} \Delta_i \\ &< 2 \sum_{i=0}^{\log \tau_{max}} \frac{1}{2^i} \Delta_i = 2c_{sync}. \end{aligned}$$

□

Our main result follows from showing that we can always find a non-decreasing solution of cost $O(\log n)$ times the optimal value. We use the following tree pairing Lemma by Klein and Ravi [26]. For this lemma, we need some additional terminology. A pairing of vertices is a collection of disjoint pairs of vertices. We say that a pairing covers an even subset S if each vertex in S is contained in one of the pairs in the pairing. Furthermore, we call the unique path between two vertices in a given tree the tree-path induced by that pair.

Lemma 11 ([26]) *Given any tree T and an even subset S of its vertices, there is a pairing of vertices covering S such that the tree-paths induced by the pairs are edge-disjoint.*

Using the tree pairing Lemma 11, we will construct non-decreasing trees to approximate arbitrary trees. First we define the notations needed for the algorithm.

A *non-decreasing* arc $a(\{u, v\})$ of $\{u, v\}$ is the arc between u and v that points from the vertex with lower turnover time to the one with higher turnover time (ties are broken arbitrarily). The vertex with the lower (higher) turnover time is denoted by $L(\{u, v\})$ ($H(\{u, v\})$). We denote by U the set of unpaired vertices and by A the set of arcs of the non-decreasing tree being constructed, and require that all arcs must eventually point away from the root. The procedure is described in Algorithm 3.

Lemma 12 *Given an arbitrary tree T of cost $c(T)$, there is a non-decreasing tree of cost at most $\lceil \log n \rceil c(T)$.*

Proof Let T be a tree. We will construct a non-decreasing tree T' by iteratively pairing the vertices and directing each pair in a non-decreasing manner. In each round, we pair a largest even subset S of U. So, if $|U|$ is even, we choose $S = U$, and if $|U|$ is odd,

Algorithm 3 Algorithm to create non-decreasing tree from arbitrary tree

```
1: Initialize $U \leftarrow V$ and $A \leftarrow \emptyset$.
2: while $|U| > 1$ do:
3:     Find an edge-disjoint pairing $P$ of a largest even subset of $U$.
4:     for $\{u, v\} \in P$ do:
5:         $A \leftarrow A \cup a(\{u, v\})$.
6:         $U \leftarrow U \setminus H(\{u, v\})$.
7:     end for
8: end while
```

we choose $S = U \setminus \{v\}$, where vertex $v \in U$ is chosen arbitrarily. By Lemma 11, there exists a pairing of vertices in S such that the tree-paths induced by the pairs are edge-disjoint. We pick an arbitrary pairing with this property. Then we direct each pair $\{u, v\}$ in ascending order of turnover times and delete the vertex with higher turnover time from U. Here, ties are broken arbitrarily. For each pair $\{u, v\}$, arc $a(\{u, v\})$ is added to the arc set of T'. The cost of arc $a(\{u, v\})$ is equal to the cost of the corresponding tree-induced path in T.

Eventually, there is only one vertex left in U. This is one of the vertices with minimum turnover time. By our construction, there is a path from this vertex to any other vertex such that the turnover times along this path are non-decreasing. Since the number of arcs we added to T' is equal to the number of vertices we deleted from U, which was initially set to be equal to V, we know that the number of arcs is equal to $n - 1$. This implies that we constructed a tree. Combining these observations shows that we have constructed a non-decreasing tree T'.

In each round, we used each edge of T at most once since all pair-induced paths were edge-disjoint. Let $\kappa(t)$ be the number of vertices in U at the beginning of round t. Hence, $\kappa(1) = n$. Since each round paired off either all vertices or all but one vertex, we have $\kappa(t) = \lceil \kappa(t-1)/2 \rceil$, for $t \geq 2$. So the total number of rounds is $\lceil \log n \rceil$. Hence $c(T') \leq \lceil \log n \rceil c(T)$. □

Proof of Theorem 9 Given an optimal solution, let T_i be the minimum Steiner tree on the set of vertices visited on day i, whence the sum of costs over all edges in the Steiner tree is no more than the cost of the tour of that day. Using Lemma 12 we can find non-decreasing trees T_i' of cost at most $O(\log n)c(T_i)$. Turning the trees into tours, by doubling the edges and traversing a depth-first search, loses only a constant factor, which gives us a non-decreasing solution of cost $O(\log n)$ times the optimal value. By Lemma 9 we may then turn this solution into a synchronized one as required, concluding the proof. □

Next, we show that our algorithm is not a constant factor approximation, by showing that there exists a class of instances where requiring a solution to be non-decreasing introduces an $\Omega(\log \log n)$ gap. Together with Lemma 10, this implies that our algorithm does no better than $O(\log \log n)$ as well.

We first show that Lemma 12 is tight. Consider the following sequence of sequences $(a^0, a^1, \ldots)$ where $a^0 = (1)$ and a^{i+1} is generated by alternatingly taking an element from a^i and then from the sequence $b^i = (2^i + 1, 2^i + 2, \ldots, 2^{i+1})$. The beginning of the sequence looks like this:

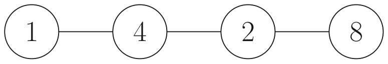

Fig. 4 Illustration of G_2 (turnover times in circles)

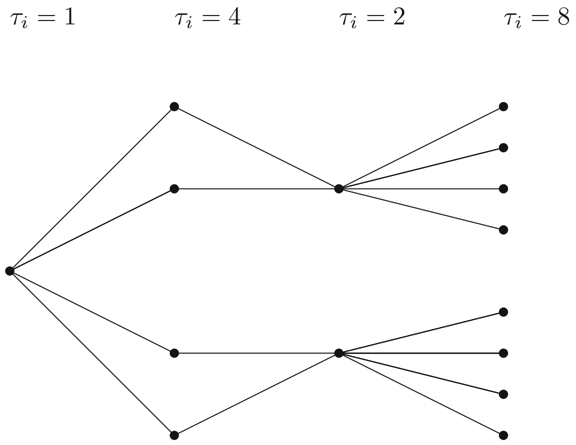

Fig. 5 Illustration of H_2

- $a^0 = 1$
- $a^1 = 1, 2$
- $a^2 = 1, 3, 2, 4$
- $a^3 = 1, 5, 3, 6, 2, 7, 4, 8$
- $\cdots$

Then define for each i the (unweighted) graph G_i as the path graph with 2^i vertices, where the jth vertex, v_j, has turnover time $2^{a^i_j - 1}$. See Fig. 4 for an example.

A minimum spanning tree in G_i, which is G_i itself, costs $2^i - 1$, since G_i contains 2^i vertices. Algorithm 3 will first choose 2^{i-1} edges of length 1. After deleting half of the vertices, it will choose 2^{i-2} edges of length 2. This repeats until only two vertices are left, which will be connected with one edge of length 2^{i-1}. Hence, the non-decreasing spanning tree produced by Algorithm 3 costs $i2^{i-1}$. Moreover, since the solution produced attaches every vertex to a nearest vertex with lower turnover time, it must be optimal.

To show our lower bound, we will define another class of graphs H_i for $i \geq 1$ that are constructed from $\{G_i\}$. The idea is to make τ_j copies of each vertex v_j, and then connect them in a regular way, for example like in Fig. 5.

Formally H_i is constructed as follows. For simplicity of description, we assume that G_i is planarly embedded from left to right, and we assume that we keep a planar embedding of H_i during construction.

We first copy vertex 1 to H_i. Now we work from left to right, starting from the second vertex. When we are at vertex v_j, we put τ_j copies of v_j vertically above each other and to the right of the copies of v_{j-1} in H_i. Then we connect them to the copies of v_{j-1} in H_i such that the graph remains planar and all copies of v_{j-1} have the same degree, and all copies of v_j have the same degree. This can be done in only one way, and it can always be done, since turnover time are powers of 2. Furthermore we identify vertex 1 with the root.

Theorem 10 *The instance induced by H_i has an $\Omega(\log\log n)$ gap between the optimal and the optimal non-decreasing solution.*

Proof There exists a solution that visits exactly one vertex of each turnover time per day, that costs $2(2^i-1)$. In this solution, we visit the first vertex with turnover time 2 on every odd day, and the other vertex with turnover time 2 on every even day. Then, we recursively apply this procedure, e.g., on days when we visit the first vertex with turnover time 2, we alternatingly visit the first and second vertex with turnover time 4.

Now suppose we impose non-decreasing constraints. In this case we need to use, for each j, one edge pointing from a vertex with turnover time less than 2^j to a vertex with turnover time 2^j at least once every 2^j days. Since there are 2^j vertices with turnover time 2^j in H_i, we need to use at least one edge, on average, pointing from a vertex with a lower turnover time to one with a higher turnover time per day. We now observe that the cost of the cheapest set of edges that contains at least one edge that enters a vertex with turnover time 2^j pointing from a vertex with lower turnover time for all j, is at least the cost of the optimal non-decreasing tree in G_i. Hence, as explained before, we get an average cost of at least $i2^{i-1}$, which is a factor $\Omega(i)$ more expensive than the optimal value. Since the number of vertices in H_i is equal to $2^{2^i}-1$, it follows that $i=\log\log(n+1)$, and the optimal solution under non-decreasing constraints is at least an $\Omega(\log\log n)$ factor more expensive than the optimal solution. □

It is an open question whether there exists a constant factor approximation algorithm for the general case. We observe that the approach of first finding a tree spanning all vertices and then using the algorithm of Sect. 3 is unsuccessful. In fact there exist instances of the problem on a graph G with n vertices, such that if we limit our attention to tours that for each day use only edges of a spanning tree of G then the obtained solution is $\Omega(\log n)$ worse. This implies that we need some new ideas, in order to improve the $O(\log n)$ approximation of the previous theorem.

Recently, Bosman and Olver have devised a new algorithm with an improved performance ratio of $O(\log\log n)$ for the MIN–AVG objective. A first version of this result is included in the PhD-thesis of Bosman [6].

5 Conclusion

In this paper, we considered replenishment problems with fixed turnover times, a natural inventory-routing problem that has not been studied before. We formally defined the RFTT problem and considered the objectives MIN–AVG and MIN–MAX. For the MIN–AVG RFTT, we showed that it is at least as hard as the intractable PINWHEEL SCHEDULING PROBLEM on series-parallel graphs and we gave a 2-approximation for trees. For the MIN–MAX objective we showed NP-hardness on stars and gave a 6-approximation for tree metrics. We also presented a DP that solved the MIN–AVG RFTT in pseudopolynomial time on line graphs. Finally, we gave an $O(\log n)$-approximation for both objectives on general metrics.

The results that we present should be considered as a first step in this area and many problems remain open. An intriguing open problem is the complexity of RFTT on a tree. Namely, for MIN–AVG variant we conjecture that the problem is hard, and we ask whether the simple 2-approximation we provide can be improved. For the MIN–MAX variant it is open whether the problem is APX-hard and whether we can improve the 6-approximation.

Other than replenishing locations with routing aspects that we studied in this paper, scheduling problems modeling maintenance or security control of systems form a class of problems to which this model naturally applies. It would also be interesting to study such fixed turnover time problems in combination with scheduling. Would this combination allow for more definitive results?

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Baller, A.C., van Ee, M., Hoogeboom, M., Stougie, L.: On the complexity of inventory routing problems when routing is easy. Networks **75**(2), 113–123 (2019)
2. Baruah, S., Goossens, J.: Scheduling real-time tasks: algorithms and complexity. In: Handbook of Scheduling: Algorithms, Models, and Performance Analysis. CRC Press, Boca Raton (2003)
3. Baruah, S., Rosier, L., Tulchinsky, I., Varvel, D.: The complexity of periodic maintenance. In: Proceedings of the International Computer Symposium, pp. 315–320 (1990)
4. Bilò, D., Gualà, L., Leucci, S., Proietti, G., Scornavacca, G.: Cutting bamboo down to size. In: Farach-Colton, M., Prencipe, G., Uehara, R. (eds.) 10th International Conference on Fun with Algorithms (FUN 2021). Leibniz International Proceedings in Informatics (LIPIcs), vol. 157, p. 5:1-5:18. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Dagstuhl (2020). https://doi.org/10.4230/LIPIcs.FUN.2021.5
5. Bonifaci, V., Marchetti-Spaccamela, A.: Feasibility analysis of sporadic real-time multiprocessor task systems. Algorithmica **63**(4), 763–780 (2012)
6. Bosman, T.: Relax, round, reformulate: near-optimal algorithms for planning problems in network design and scheduling. Ph.D. thesis, Vrije Universiteit Amsterdam (2019)
7. Bosman, T., van Ee, M., Jiao, Y., Marchetti-Spaccamela, A., Ravi, R., Stougie, L.: Approximation algorithms for replenishment problems with fixed turnover times. In: Bender, M., Farach-Colton, M., Mosteiro, M.A. (eds.) LATIN 2018: Theoretical Informatics. Lecture Notes in Computer Science, vol. 10807, pp. 217–230. Springer, New York (2018)
8. Calabro, C., Impagliazzo, R., Kabenets, V., Paturi, R.: The complexity of unique k-sat: An isolation lemma for k-cnfs. J. Comput. Syst. Sci. **74**(3), 386–393 (2008)
9. Chan, M.Y., Chin, F.Y.L.: General schedulers for the pinwheel problem based on double-integer reduction. IEEE Trans. Comput. **41**(6), 755–768 (1992)
10. Chan, M.Y., Chin, F.Y.L.: Schedulers for larger classes of pinwheel instances. Algorithmica **9**(5), 425–462 (1993)
11. Chuangpishit, H., Czyzowicz, J., Gasieniec, L., Georgiou, K., Jurdzinski, T., Kranakis, E.: Patrolling a path connecting a set of points with unbalanced frequencies of visits. In: Tjoa, A.M., Bellatreche, L.,

Biffl, S., van Leeuwen, J., Wiedermann, J. (eds.) SOFSEM 2018: Theory and Practice of Computer Science, pp. 367–380. Springer, Cham (2018)
12. Coelho, L.C., Cordeau, J.F., Laporte, G.: Thirty years of inventory routing. Transp. Sci. **48**(1), 1–19 (2013)
13. Coene, S., Spieksma, F.C.R., Woeginger, G.J.: Charlemagne's challenge: the periodic latency problem. Oper. Res. **59**(3), 674–683 (2011)
14. Damaschke, P.: Two robots patrolling on a line: integer version and approximability. In: Gasieniec, L., Klasing, R., Radzik, T. (eds.) Combinatorial Algorithms, pp. 211–223. Springer, Cham (2020)
15. Dell, H., Husfeldt, T., Marx, D., Taslaman, N., Wahlén, M.: Exponential time complexity of the permanent and the tutte polynomial. ACM Trans. Algorithms **10**(4), 211–2132 (2014)
16. Della Croce, F.: An enhanced pinwheel algorithm for the bamboo garden trimming problem. arXiv preprint arXiv:2003.12460 (2020)
17. Eisenbrand, F., Hähnle, N., Niemeier, M., Skutella, M., Verschae, J., Wiese, A.: Scheduling periodic tasks in a hard real-time environment. In: Proceedings of the 37th International Colloquium on Automata, Languages, and Programming, pp. 299–311. Springer (2010)
18. Ekberg, P., Yi, W.: Schedulability analysis of a graph-based task model for mixed-criticality systems. Real-Time Syst. **52**(1), 1–37 (2016). https://doi.org/10.1007/s11241-015-9225-0
19. Fakcharoenphol, J., Rao, S., Talwar, K.: A tight bound on approximating arbitrary metrics by tree metrics. J. Comput. Syst. Sci. **69**(3), 485–497 (2004)
20. Fishburn, P.C., Lagarias, J.C.: Pinwheel scheduling: achievable densities. Algorithmica **34**(1), 14–38 (2002)
21. Frederickson, G.N., Hecht, M.S., Kim, C.E.: Approximation algorithms for some routing problems. In: Proceedings of the 17th International Symposium on Foundations of Computer Science, pp. 216–227 (1976)
22. Garey, M.R., Johnson, D.S.: Complexity results for multiprocessor scheduling under resource constraints. SIAM J. Comput. **4**(4), 397–411 (1975)
23. Gasieniec, L., Klasing, R., Levcopoulos, C., Lingas, A., Min, J., Radzik, T.: Bamboo garden trimming problem. In: SOFSEM, pp. 229–240. Springer (2017)
24. Holte, R., Mok, A., Rosier, L., Tulchinsky, I., Varvel, D.: The pinwheel: a real-time scheduling problem. In: Proceedings of the 22th Annual Hawaii International Conference on System Sciences, vol. 2, pp. 693–702 (1989)
25. Jacobs, T., Longo, S.: A new perspective on the windows scheduling problem. arXiv preprint arXiv:1410.7237 (2014)
26. Klein, P., Ravi, R.: A nearly best-possible approximation algorithm for node-weighted steiner trees. J. Algorithms **19**(1), 104–115 (1995)
27. Mok, A., Rosier, L., Tulchinksy, I., Varvel, D.: Algorithms and complexity of the periodic maintenance problem. Microprocess. Microprogram. **27**(1–5), 657–664 (1989)
28. van Ee, M.: A 12/7-approximation algorithm for the discrete bamboo garden trimming problem. Oper. Res. Lett. **49**(5), 645–649 (2021)

Authors and Affiliations

Thomas Bosman[1] · Martijn van Ee[2] · Yang Jiao[3] · Alberto Marchetti-Spaccamela[4,5] · R. Ravi[6] · Leen Stougie[5,7,8]

Thomas Bosman
tbosman@gmail.com

Yang Jiao
yang.jiao@boeing.com

Alberto Marchetti-Spaccamela
alberto@diag.uniroma1.it

R. Ravi
ravi@andrew.cmu.edu

Leen Stougie
leen.stougie@cwi.nl

1 Amsterdam, The Netherlands

2 Netherlands Defence Academy, Den Helder, The Netherlands

3 Boeing Research & Technology, Tukwila, WA, USA

4 Sapienza University of Rome, Rome, Italy

5 Erable, INRIA, Lyon, France

6 Tepper School of Business, Carnegie Mellon University, Pittsburgh, PA, USA

7 Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands

8 Vrije Universiteit, Amsterdam, The Netherlands