Online Algorithms for the Linear Tape Scheduling Problem

Carlos Cardonha, Lucas C. Villa Real
IBM Research
Rua Tutoia 1157, Vila Mariana
Sao Paulo, SP, Brazil

Abstract

Even in today's world of increasingly faster storage technologies, magnetic tapes continue to play an essential role in the market. Yet, they are often overlooked in the literature, despite the many changes made to the underlying tape architecture since they were conceived. In this article, we introduce the LINEAR TAPE SCHEDULING PROBLEM (**LTSP**), which aims to identify scheduling strategies for *read* and *write* operations in single-tracked magnetic tapes that minimize the overall response times for read requests. Structurally, LTSP has many similarities with versions of the Travelling Repairman Problem and of the Dial-a-Ride Problem restricted to the real line. We investigate several properties of LTSP and show how they can be explored in the design of algorithms for the online version of the problem. Computational experiments show that the resulting strategies deliver very satisfactory scheduling plans, which in most cases are clearly superior (potentially differing by one order of magnitude) to those produced by a strategy currently used in the industry.

Introduction

Magnetic tapes play a significant role as a medium for storage of digital data. Even though they frequently go unnoticed, the industry continues to adopt them due to their remarkable cost-effectiveness; price per bit ratios in magnetic tapes are still typically smaller than those offered by other technologies (Fontana, Decad, and Hetzler 2013). Additionally, tapes are optimized for streaming operations; in particular, sequential data transfer speeds are relatively high (around 300MB/second in some scenarios), which makes tapes specially suitable for several classes of applications as e.g., audio/video streaming, automated backups, and long-term archiving.

The demand for long-term archiving of digital data can be easily observed in the media and entertainment industry, where high-resolution digital content produced often needs to be stored for eventual accesses in the future (e.g., a replay of a soccer game); frequently that content remains archived for many years without ever being read (Frank et al. 2012). Another instance happens in the oil industry, in which sonar systems aboard vessels routinely capture more data from deep oceans than the available computers can process, resulting in the archival of that data for eventual processing – akin to situations faced by large scientific organizations (Murray et al. 2012) and renowned projects such as the Large Hadron Collider (Cavalli et al. 2010). Finally, we remark that tapes are key components of distributed computing-based storage solutions (Prakash et al. 2013) and are used for online reading and writing of data files (Real et al. 2015) just like more popular removable media types (Pease et al. 2010).

Operationally, a modern magnetic tape works as follows. Since the reading/writing head is in a fixed position, the tape drive first needs to rewind or fast-forward the tape to the requested location. Next, the head needs to be moved up or down to select the correct track before the requested operations can execute. Because access in tapes is sequential, the response times for write and read requests depend strongly on the tape position at the head and on the order in which operations are executed (Sandstå and Midtstraum 1999). We also note that, differently from read operations, writes are often handled asynchronously by storage systems through the use of caching memories. As a consequence, readers may experience higher response times from the tape drive than writers, and thus need to be processed with special care to prevent task starvation.

In this work, we investigate challenges associated with the scheduling of *read* and *write* requests in magnetic tapes. Since data stored within a partition of a modern enterprise tape cannot be updated without causing data loss to the tracks nearby, we do not consider scenarios where files can be overwritten, modified, or removed from the tape. Writes are always handled at the end of the tape through append operations, as in *Write-Once, Read-Many* storage devices (Hasan et al. 2005).

Limited computational power and rigorous time restrictions prohibit the applicability of complex (i.e., time-consuming) algorithms in the practical settings. Namely, scheduling plans are expected to be produced in less than one second (two hundred milliseconds is the rule-of-thumb), a limitation that hinders the application of traditional operational research techniques such as mixed-integer linear programming and constraint programming. For this reason, most techniques reported on related work simply sort the list of requests according to their offset relative to the current position of the tape (Schaeffer and Casanova 2011;

Zhang et al. 2006). More complex approaches incorporate the cost of repositioning the tape on different tape models, but they rely on time-consuming characterizations of the physical tapes and on access to low level hardware information (Sandstå and Midtstraum 1999; Hillyer and Silberschatz 1996).

We restrict our attention in this work to scenarios where the tape has a single track, that is, it moves only to the left and to the right with the reading/writing head fixed at a certain vertical offset. Although strong, this assumption reflects situations where batches of requests are relatively "local" (i.e., associated with files which are part of the same working set, thus close to each other in the tape). Moreover, one can find in the literature tape partitioning schemes to enable better spatial arrangements of groups of files (Oracle 2011) and that support the scenarios we describe. Additionally, we assume that the velocity with which the tape is moved is always constant; in practice, tape acceleration and momentum do have an impact on the velocity in which the request is served, though, so this implies in a simplification of the real problem. The main contributions of this article are the introduction of the LINEAR TAPE SCHEDULING PROBLEM, the study of its structural properties, and the algorithms we propose for the online version of the problem.

The article is organized as follows. First, we present a list of related problems that have been studied already in the literature. Then, after providing a formal description of the problem, we investigate some algorithms and properties of the offline LTSP. Afterwards, we investigate structural properties of the online LTSP and present five algorithms for the problem. Finally, we report on the results of computational experiments involving the algorithms for the online LTSP and finish the article with our conclusions.

Background and related work

The LINEAR TAPE SCHEDULING PROBLEM is related to the TRAVELLING REPAIRMAN PROBLEM (**TRP**), a variant of the TRAVELLING SALESMAN PROBLEM (**TSP**) in which requests are generated on (a subset of) the vertices of a graph and the goal is to minimize the sum of their waiting times (Afrati et al. 1986). The **TRP** has been largely studied in the literature (eventually with different names, such as DELIVERY MAN PROBLEM or MINIMUM LATENCY PROBLEM). Afrati et al. showed that the **TRP** is NP-complete in the general case and that the *line*-**TRP**—a version of the problem in which all requests are distributed in a line—can be solved in polynomial time if requests do not have deadlines and their servicing times are insignificant (or are equal for all machines) (Afrati et al. 1986). Recently, Bock presented the state-of-the-art of the *line*-**TRP** with respect to the computational complexity of its various versions (and of the *line*-**TSP** as well, in which the goal is to minimize the tour duration) (Bock 2015); in particular, the complexity of the *line*-**TRP** without time-windows (i.e., without release times and due dates) and with general processing times is still unknown.

The LINEAR TAPE SCHEDULING PROBLEM is also related to the SINGLE-VEHICLE SCHEDULING PROBLEM ON A LINE (**LINE-VSP** or **VSP-PATH**); in this problem, the goal is to minimize the completion time of jobs associated with vertices, while eventually taking into account elements such as release time, deadlines, handling (or service) time, and initial/final position of the vehicle. The version with handling times and time windows is NP-complete (Garey and Johnson 1979). In scenarios where service times are equal to zero and only release times are considered, the problem can be solved in polynomial time (Psaraftis et al. 1990). Karuno, Nagamochi, and Ibaraki presented a polynomial-time 1.5 approximation algorithm for **LINE-VSP** with release and handling times (Karuno, Nagamochi, and Ibaraki 1998); additionally, Karuno and Nagamochi showed that the same problem admits a Polyonomial-Time Approximation Scheme (PTAS) if the number of vehicles is fixed and larger than one (Karuno and Nagamochi 2001). For other related results, see e.g. Tsitsiklis (Tsitsiklis 1992) and Karuno and Nagamochi (Karuno and Nagamochi 2005).

The DIAL-A-RIDE PROBLEM is also similar to the **LTSP**; in this problem, fleets of vehicles are used to transport products between vertices in a graph. De Paepe et al. provide an extensive classification of Dial-a-Ride problems (Paepe et al. 2004); in particular, they show that the minimization of the sum of all completion times is NP-complete even in scenarios where there is just one vehicle of capacity one (i.e., can only carry one product at any point in time) servicing all requests and all vertices are positioned in a line (i.e., $\mathbb{R}^1$).

In practical settings, the LINEAR TAPE SCHEDULING PROBLEM is inherently online, that is, decisions are taken (and executed) in real-time, before all requests become available. Moreover, there is no fixed execution order for read requests, thus making our problem related to the ONLINE TRAVELLING SALESMAN PROBLEM (**OLTSP**), a variation of the TRAVELLING SALESMAN PROBLEM in which new vertices to be visited are informed to the salesman during the tour. In particular, the **LTSP** is related to the *Homing*-**OLTSP**, defined by Ausiello et al. (Ausiello et al. 2001), since the end of the tape needs to be sought to from time to time in order to execute writing requests. In the same work, the authors present a 1.75-competitive algorithm for the *Homing*-**OLTSP** in a line and show that no online algorithm can be better than 1.64-competitive for the problem. Handling times are zero for the *Homing*-**OLTSP**, but Augustine shows that any c-competitive online algorithm for the problem with zero handling times yields a $(c+1)$-competitive online algorithm for non-negative handling times (Augustine 2002).

Problem description

Let T be a single-track magnetic tape with infinite (or sufficiently large) storage capacity. Each operation taking place on T is the response to one or more *jobs* (or *requests*) associated with exactly one *file* f; the set of jobs is denoted by $\mathcal{J}$, the set of files stored in T is denoted by $\mathcal{F}$, and $J(f) \subseteq \mathcal{J}$ is the set of jobs associated with f. Each job j in $\mathcal{J}$ has a *type*, given by function $t : \mathcal{J} \to \{r, w\}$; $t(j) = w$ if j is a *write request*, and $t(j) = r$ if j is a *read request*. The file associated with job j is given by $f(j)$.

All operations take place from the left to the right. That is, in order to execute all currently pending requests associated

with some file f of *size* $s(f)$, the tape drive's head needs to move from f's *leftmost bit*, denoted by $l(f)$, to f's *rightmost bit*, denoted by $r(f)$ and given by $r(f) = l(f) + s(f)$. We assume that different files may have different sizes. For $f_1, f_2 \in \mathcal{F}$, we say that f_1 is on the *left* of f_2 if $l(f_1) < l(f_2)$. For write requests, the head needs to be on the leftmost unoccupied bit of the tape (or, more informally, in the *end of* T), denoted by m. The velocity with which the head moves is constant and independent from the execution of operations (i.e., the velocity does not change if read and write operations are executed during the movement).

Because each job is associated with exactly one file (whereas several jobs may be associated with the same file), we overload notation and apply the functions defined in $\mathcal{F}$ to jobs in $\mathcal{J}$ (referring obviously to the associated files) without risks of ambiguity. A *pending request* is a request that has not been serviced yet; the number of pending requests associated with file f is denoted by $n(f)$.

For any file f, let $\mathcal{J}(f) \subset \mathcal{J} \setminus J(f)$ be the set of pending requests which are *not* associated with f. Moreover, $\mathcal{J}^L(f) \subset \mathcal{J}(f)$ and $\mathcal{J}^R(f) \subset \mathcal{J}(f)$ are the subsets of $\mathcal{J}(f)$ containing all pending requests associated with files located to the left and to the right of $l(f)$, respectively.

Write requests are executed according to a FIFO policy and their response times are irrelevant; namely, the execution of a write job becomes urgent only after the arrival of a read job associated with the same file. In order to explore this aspect, tape systems use a *buffer area* to temporarily store file chunks associated with write requests, postponing hence the execution of these requests. We assume in this work that the buffer has infinite capacity. Finally, there is no enforced policy dictating how read requests should be processed, and this is the point where a scheduling strategy plays a role in LTSP. Informally, we can say that LTSP is about the identification of scheduling plans that deliver low *overall response times* (the sum of all response times) to read requests; for example, if two read requests associated with some file f are released at times 1 and 5 and f starts to move from $l(f)$ to $r(f)$ at time 10, the response time will be equal to $10-1 = 9$ for the first request and to $10 - 5 = 5$ for the second one, leading thus to an increment of 14 on the cost (or objective value) of the associated scheduling plan. Formally, LTSP is defined as follows:

Definition 1 (LINEAR TAPE SCHEDULING PROBLEM - LTSP)**.** *Given a set of files $\mathcal{F}$, a set of jobs $\mathcal{J}$, a type function t, and a size function s, schedule all jobs in $\mathcal{J}$ in a way that the sum of response times for read requests is minimized.*

Theoretical and algorithmic aspects

Real-world scenarios of LTSP are actually online optimization problems, since scheduling decisions must be taken in situations where knowledge about $\mathcal{J}$ might still be incomplete (that is, decisions must be taken before all jobs are released). Nevertheless, online problems frequently share structural properties with their offline counterparts, and LTSP is no exception to this rule. Therefore, in this section, we investigate theoretical and algorithmic aspects of the offline and of the online versions of LTSP.

The Offline LTSP

The offline LTSP is the version of the problem in which the head is initially positioned at the end of T, $\mathcal{J}$ contains only read requests, and all jobs are released at time 0. One important characteristic of LTSP is the fact that execution times and tape movements are "merged", since a read operation in a file necessarily takes the tape to the leftmost bit of the next file. Moreover, after this movement, nothing can be assumed about the direction it should move afterwards; in some cases the tape should keep moving to the right (in order to service other requests and/or to return to the end of the tape), and in other cases it should move backwards. This aspect leads to a subtle but significant difference between the offline LTSP and the Travelling Repairman Problem in a line with general processing times and no time windows (that is, without release times and due dates for jobs): note that, in the latter, the repairman's position will not change from the time she visits a node to the time she is done repairing the entity that node represents. The complexity of this version of the TRP is still unknown.

The offline LTSP is equivalent to the special case of the Dial-a-Ride Problem in a line with a single vehicle in which the trajectories described by each route are pairwise non-intersecting, the sources are always on the left of their associated destinations, and the vehicle starts to the right of the right-most destination. The reduction employed by de Paepe et al. (based on the Circular Arc Colouring Problem) cannot be used in this special case, leaving hence the complexity of this problem open (Paepe et al. 2004).

We were not able to identify the complexity of the offline LTSP, but we present below some identities and properties that have been explored in the design of the algorithms proposed in this work. First, we observe that optimal scheduling plans for instances of the offline LTSP can be divided in two phases. In *Phase 1*, the tape is sought from its end (position m) to its beginning (position 1), i.e., from the right to the left. Phase 1 may be intercalated by one or more *mini-batches*; a mini-batch b is composed of an *execution movement* (or *execution phase*) from $l(b)$ to $r(b)$ and a *return movement* (or *return phase*) from $r(b)$ to $l(b)$. A mini-batch b' cannot be executed between the end of the execution movement and the start of the return movement of some other mini-batch b; in these situations, we say that all movements are associated with the same mini-batch. We extend the notation used for files and jobs to mini-batches; that is, a mini-batch b has a size $s(b)$, a leftmost bit $l(b)$, and a rightmost bit $r(b)$, with $s(b) = r(b) - l(b)$. If $l(b)$ and $r(b)$ coincide with the leftmost and the rightmost bits of some file in $\mathcal{F}$, we say that b is an *atomic mini-batch*.

In *Phase 2*, the head of T moves from position 1 to position m, servicing all the remaining pending requests (i.e., those not executed during Phase 1). In any optimal solution for LTSP, all pending jobs associated with file f are serviced (simultaneously) whenever f is traversed from the left to the right, so we incorporate this behaviour in all algorithms presented in the article. Moreover, as a consequence, it follows that the tape never moves to the left in *Phase 2*.

We assume in this subsection w.l.o.g. that the leftmost file is associated with at least one request in $\mathcal{J}$. If this is not

the case, the movement of the head to position 1 should be substituted for a movement to the leftmost bit of the leftmost file associated with a request in $\mathcal{J}$. The following results hold for the offline LTSP:

Observation 1. *In any optimal scheduling plan for the offline LTSP, every mini-batch b is such that $l(b)$ and $r(b)$ coincide with the leftmost and rightmost bits of (not necessarily distinct) files in $\mathcal{F}$, respectively.*

Proposition 1. *In any optimal scheduling plan for the offline LTSP, the executions of any pair of mini-batches do not intersect in time.*

Proof. Let b and b' be mini-batches such that $l(b) \leq l(b')$. Their execution intersect in time if b' starts before b is finished, so we have two cases to analyse.

If b' starts during the return phase of b, we have $l(b') \in [l(b), r(b)]$. Moreover, the tape head traverses region $[l(b'), \min(r(b), r(b')]$ three times and, in particular, the second movement from the left to the right clearly increases the response time of pending requests unnecessarily, since it does not involve the execution of any pending job. Under these circumstances, a plan employing b and b' can be improved if this pair of mini-batches is substituted for a single mini-batch b^* such that $l(b^*) = l(b)$ and $r(b^*) = \max(r(b), r(b'))$.

Finally, if b' starts during the execution phase of b, then the behaviour of the head will be similar to the one resulting from the substitution of b and b' for b^*. Therefore, we conclude that mini-batches do not intersect in time in optimal solutions for the offline LTSP. □

Proposition 2. *In any optimal scheduling plan for the offline LTSP, every mini-batch starts from a position that is being visited for the first time.*

Proof. Let P be an optimal scheduling plan containing a mini-batch b such that $l(b)$ is not being traversed for the first time; moreover, let $p' < l(b)$ be the leftmost bit that has been visited by the time b will start.

In the offline LTSP, the head of T visits a position for the second time either during the execution of a mini-batch or in Phase 2. Since mini-batches do not take place during Phase 2, this second visit is associated with the execution of a mini-batch whose leftmost bit is p'. However, it follows from the definition of mini-batches and from Proposition 1 that two mini-batches do not intersect in time in any optimal scheduling plan for the offline LTSP, contradicting therefore the optimality of P. □

Observe that Propositions 1 and 2 do not necessarily hold in online settings, since modifications in a mini-batch might be motivated by the release of new requests.

The minimization of the makespan (time when all jobs are serviced) is trivial for the offline LTSP; an optimal solution for the problem with this optimization criteria consists of moving the head from position m directly to position 1, withouth mini-batches, and then returning to the end of the tape while servicing all requests. Clearly, the same algorithm also minimizes the distance traversed on tape. Karuno, Nagamochi, and Ibaraki showed that this strategy, to which they refer as *simple schedule*, yields a 1.5-approximation algorithm for the Single-Vehicle Scheduling Problem on a line with release and handling times (Karuno, Nagamochi, and Ibaraki 1998). The proposition below shows that this strategy may deliver arbitrarily bad plans for the offline LTSP.

Proposition 3. *The simple schedule strategy is not a c-approximation for the offline LTSP for any constant c.*

Proof. Let m be a perfect square integer. Let $\mathcal{F} = \{f_1, f_2, \ldots, f_{\sqrt{m}}\}$ be such that $l(f_1) = s(f_1) = 1$, $l(f_2) = 2$, $s(f_2) = m - \sqrt{m}$, $l(f_i) = m - \sqrt{m} + i - 1$ and $s(f_i) = 1$ for i in $[3, \sqrt{m}]$, and assume that $\mathcal{J}$ contains exactly one read request associated with each file in $\mathcal{F} \setminus \{f_2\}$. According to the simple schedule strategy, the head moves to position 1 and returns to position m, executing $f_1, f_2, \ldots, f_{\sqrt{m}}$ in the order defined by their indices. Asymptotically, the resulting overall request time yielded by this strategy is

$$m\sqrt{m} + O(\sqrt{m}(m - \sqrt{m})) + O\left(\frac{\sqrt{m}\sqrt{m}}{2}\right);$$

first, all $\sqrt{m}$ jobs wait for the head to move from position m to 1 (resulting in a penalty of $m\sqrt{m}$); then, after the execution of $J(f_1)$, the remaining $O(\sqrt{m})$ pending jobs wait for the head to move forward $O(m - \sqrt{m})$ positions (resulting in a penalty of $O(m\sqrt{m} - m)$); finally, these $O(\sqrt{m})$ jobs are serviced sequentially (with aggregate penalty given by $\sqrt{m} + (\sqrt{m} - 1) + \ldots + 1 = O(\frac{\sqrt{m}\sqrt{m}}{2})$), resulting in an overall response time of $O(m\sqrt{m})$.

Alternatively, if a mini-batch b such that $l(b) = m - \sqrt{m} + 2$ and $r(b) = m$ is performed in Phase 1, leaving only the job associated with f_1 to be executed during Phase 2, the resulting overall request time is asymptotically equal to

$$\sqrt{m}\sqrt{m} + \frac{\sqrt{m}\sqrt{m}}{2} + m,$$

which is $O(m)$. The ratio between the penalties of both strategies for this family of scenarios converges asymptotically to $O(\sqrt{m})$, and since m can be made arbitrary large, we conclude that the simple schedule strategy is not c-competitive for the offline LTSP for any constant c. □

A second strategy consists of having one atomic mini-batch in Phase 1 for each file f in the tape such that $n(f) > 0$ (except for the file in the first position of T, which is left for Phase 2). We refer to this as the *greedy strategy*, and the proposition below shows that it is better than the simple schedule strategy with respect to worst-case scenarios.

Proposition 4. *The greedy strategy is a 3-approximation for the offline LTSP.*

Proof. For any instance of the offline LTSP, a lower bound for the response time of any job j in $\mathcal{J}$ is given by

$$m - l(j) = z(j) + \sum_{\substack{f' \in \mathcal{F} \\ l(f') > l(j) \text{ and } n(f') > 0}} s(f'); \qquad (1)$$

the left-hand side of Equality 1 shows clearly that this lower bound is equal to the distance between the leftmost bit of

file $f(j)$ and the end of the tape. This value is decomposed on the right-hand side in two parts; the sum of the sizes of all files associated with pending requests in $\mathcal{J}^R(j)$ and $z(j) \geq 0$, which represents the region of the tape containing files that have not been requested but need to be nevertheless traversed for $f(j)$ to be reached.

In the greedy strategy, all requests in $\mathcal{J}^R(j)$ are serviced in atomic mini-batches before $l(j)$ is reached; consequently, the response time of j in the resulting plan is given by

$$z(j) + \sum_{\substack{f' \in \mathcal{F} \\ l(f') > l(j) \text{ and } n(f') > 0}} 3s(f'), \tag{2}$$

since the head of the tape traverses each file associated with jobs in $\mathcal{J}^R(j)$ exactly three times before moving to $l(j)$. These observations hold for every $j \in \mathcal{J}$, and since $z(j) \geq 0$, it follows that the greedy strategy is a 3-approximation for the offline LTSP. Below, we show that this factor is tight.

Let us assume that $|\mathcal{F}| = 2$, with $l(f_1) = s(f_1) = 1$, $l(f_2) = 2$, $s(f_2) = m - 1$, $n(f_1) = m - 1$, and $n(f_2) = 1$, that is, there are two files in T, with $m - 1$ requests associated with the first and one with the second. The overall request time delivered by the greedy strategy for this family of scenarios is $m(m-1)+2(m-1)^2+1 = 3m^2-5m-1$, whereas the plan delivered by the simple schedule strategy yields an overall response time of m^2+1; asymptotically, the ratio between both solutions converge to 3, showing hence that this approximation rate is tight for the offline LTSP. □

The online LTSP

In the online LTSP, release times of jobs are unknown and $\mathcal{J}$ contains both write and read requests. Moreover, preemption is not allowed, that is, once a file associated with pending jobs starts to be traversed, its end must be reached. Scheduling plans for the online LTSP consist of *batches*, and each batch is composed of one Writing Phase, one Phase 1, and one Phase 2. As the name suggests, pending writing requests are moved from the buffer and executed during the Writing Phase. Phase 1 and Phase 2 have the same behaviour we described for the offline LTSP. The proposition below shows that the online LTSP is hard from the theoretical standpoint.

Proposition 5. *There is no algorithm able to produce c-competitive solutions for the online LTSP for any constant c even if preemption is allowed.*

Proof. Let A be an arbitrary algorithm for the online LTSP, and let us consider a scenario in which $\mathcal{F} = \{f_1, f_2\}$, with $l(f_1) = 1$, $s(f_1) = k$, $l(f_2) = k + 1$, and $s(f_2) = k^2$. As $m = k^2 + k$, it follows that the amount of time the head of the tape needs to reach any position in the tape is $O(k^2)$. Let us suppose that some job j is released with $f(j) = f_2$.

If A consumes time $\omega(k^2)$ to start servicing j, no more jobs are released; in this case, an optimal algorithm start servicing j in time $O(k^2)$, yielding thus a performance ratio for A which is not bounded by any constant value.

Conversely, if A starts traversing f_2 in time $O(k^2)$, k jobs associated with f_1 are released as soon as the head of the tape reaches position $r(f_2)$ after servicing f_2. In this case, the overall response time produced by A would be $O(O(k^2) + k(k^2 + k)) = O(k^3)$, whereas an optimal solution would have moved the head of T to $l(f_1)$ first and from there to $r(f_2)$, with an overall response time of $O(k^2) + k = O(k^2)$. Again, the ratio between the solution of A and the optimal is not bounded by any constant value.

Since no assumption was made about A, we conclude that there is no algorithm able to produce c-competitive solutions for the online LTSP for any constant c. Finally, the same constructions and results are achieved even if preemption is allowed, so the result extends to these scenarios as well. □

Proposition 5 is based on competitive analysis, which typically relies on worst-case scenarios that might not be common in practice. Below, we discuss aspects of the problem that will be explored in the scheduling strategies presented in the next section.

First, observe that the execution of a mini-batch b extends the response time of all requests in $\mathcal{J}(b)$ by $2s(b)$. Moreover, it follows directly from Proposition 2 that files in $\mathcal{J}^L(b)$ have not been visited in the current batch by the time $l(b)$ is visited for the first time, so $\mathcal{J}^L(b) = \{j \in \mathcal{J} : l(j) < l(b)\}$ and the extension in response times is not smaller than $s(b)|\mathcal{J}^L(b)|$.

Conversely, if the execution of job j is postponed to Phase 2, its response time increases by some value in $[2l(j), 2l(j)^2]$ from the time $l(j)$ has been visited for the first time in the current batch. The lower bound follows from the fact that $2l(j)$ is the minimum amount of time the head needs to go from $l(j)$ to position 1 and back. The response time increases if mini-batches starting at files associated with jobs in $\mathcal{J}^L(j)$ are executed during the Phase 1 of this batch; in the worst case, $O(l(j))$ mini-batches of length $O(l(j))$ are executed, yielding thus an upper bound of $2l(j)^2$.

Proposition 6. *For any instance of offline LTSP, jobs associated with file f should be serviced in Phase 1 if*

$$l(f)n(f) > s(f)(|\mathcal{J}^L(f)|).$$

Proof. If $l(f)n(f) > s(f)(|\mathcal{J}^L(f)|)$ and f is left to be executed in Phase 2, the resulting penalty on $J(f)$ will be larger than the penalty associated with the waiting time of all requests in $\mathcal{J}^L(f)$ if f were included in a mini-batch; analogously, any scheduling plan which postpones the execution of f to Phase 2 can be improved with the inclusion of a mini-batch b such that $l(b) = l(f)$ and $r(b) = r(f)$. □

Proposition 6 is a formalization of the following intuitive characteristics of the problem; short files and files with a large number of associated requests should be visited in Phase 1. Observe that, if the inequality is not valid, though, no conclusion can be reached if the time spent by the head to reach the first position of the tape is unknown. A similar argument can be used to show when the read operation on a certain file should be postponed to Phase 2.

Proposition 7. *For any instance of the offline LTSP, jobs associated with file f should be serviced in Phase 2 if*

$$l(f)^2 n(f) < s(f)|\mathcal{J}^L(f)|.$$

Solutions with non-atomic mini-batches may yield better overall response times than those containing only atomic mini-batches; this phenomenon can be observed in the family of instances presented in the proof of Proposition 3, for example. In general, given two mini-batches b_1 and b_2 such that $r(b_1) < l(b_2)$, having both merged on a unique mini-batch leads to a smaller overall response time if

$$s(b_2)n(b_1) > (s(b_1) + l(b_2) - r(b_1))n(b_2) + |\mathcal{J}(b_1) \cap \mathcal{J}(b_2)|(l(b_2) - r(b_1)), \quad (3)$$

that is, if the penalty on the $n(b_1)$ tasks covered by b_1 caused by the execution of b_2 is larger than the additional penalty on the $n(b_2)$ tasks covered by b_2 and on all tasks in $\mathcal{J}(b_1) \cap \mathcal{J}(b_2)$. Observe that this bound is not tight if there are pending requests associated with files located between $r(b_1)$ and $l(b_2)$; in this case, the benefits of merging b_1 with b_2 increases, as these requests would be serviced earlier.

Algorithms for the online LTSP

The algorithms for the online LTSP presented in this work are *adaptive*, since they incorporate jobs "on-the-fly", that is, they do not wait for the end of the current batch; the update of $\mathcal{J}$ with new requests is indicated in the pseudo-codes by operation $\mathbf{update}(\mathcal{J})$. During the execution of the algorithms, the current position of the head is denoted by h (initialized as $h = m$); $f(h)$ is the file over which the head is located. Operation $\mathbf{MiniBatch}(\mathcal{F}')$ represents the execution of a mini-batch on the files belonging to set $\mathcal{F}' \subseteq \mathcal{F}$. More precisely, the head moves from the leftmost bit of the leftmost file to the rightmost bit of the rightmost file in $\mathcal{F}'$.

Operation $\mathbf{WritePhase}(\mathcal{J})$ is presented in Algorithm 1 and is used for the execution of all pending write requests in $\mathcal{J}$. This phase takes place in the beginning of each batch if and only if there are pending write requests in $\mathcal{J}$, and requires h to be placed in the last position of T.

Algorithm 1 WRITEPHASE

```
procedure WritePhase(J)
    while {j ∈ J : t(j) = w} ≠ ∅ do
        select first incoming job j such that t(j) = w
        write j on end of tape
        m := m + s(j)                          ▷ update end of tape
        h := m
        F := F ∪ {f(j)}                        ▷ update set of files
        J := J \ {j}                           ▷ remove write request j from J
        update(J)
```

Algorithm 2 is the pseudo-code for Phase 2 and represents the movement of the head towards the end of the tape. Phase 2 is finished earlier if some file f is reached such that $\mathcal{J}^R(f) = 0$, that is, if there are no pending requests associated with files located to the right of f; note that h necessarily reaches m if there are pending write requests in $\mathcal{J}$.

Algorithm 3 contains the pseudo-code for *full greedy strategy* (FGS), the online extension of the greedy strategy presented for the offline LTSP. In each batch, an atomic mini-batch is executed for each file associated with some job in $\mathcal{J}$.

Algorithm 2 PHASE2

```
▷ Move head from the beginning to the end of tape
procedure Phase2(J)
    ▷ Stop if there are no more requests to the right
    while J^R(f(h)) > 0 do
        h := r(f(h))
        J := J \ J(f)                          ▷ execute all requests in J(f)
        update(J)
```

Algorithm 3 FULL GREEDY STRATEGY (FGS)

```
while J ≠ ∅ do
    WritePhase(J)
    ▷ Stop if there are no more requests to the left
    while J^L(f(h)) > 0 do
        f := f(h)
        h := l(f)
        if n(f) ≠ 0 then                       ▷ File f has pending job
            MiniBatch({f})
            J := J \ J(f)
        update(J)
    Phase2(J)
```

Algorithm 4 contains the pseudo-code for the *unique queue strategy* (UQS), a procedure that is currently being used in the magnetic tape industry. This algorithm employs a FIFO (first-in, first-out) strategy, that is, the order in which jobs are serviced is defined by their *release times*, regardless of their type; for this reason, UQS does not have its operations divided into phases. All pending requests in $J(f(j))$ are serviced together with j, and whenever the head moves to the right, all requests associated with files traversed during this movement are also executed. UQS moves the head to the left only when it needs to traverse a file positioned to the left of its current position, so it does not execute mini-batches. Finally, if $\mathcal{J}$ becomes empty, the head stands still until a new request appears.

The inequalities derived previously are valid for the offline LTSP, but we are going to employ them here in order to create variations of the full greedy strategy. The first one, denoted *selective greedy strategy* (SGS), is described in Algorithm 5 and employs Proposition 6 in order to select files to be executed as atomic mini-batches during Phase 1.

The *merging greedy strategy* (MGS), described in Algorithm 6, has two differences from SGS. First, it estimates the amount of time $d(f)$ the head will need to move from $l(f)$ to the first position of the tape and back assuming that all files will be subject to the verification of Proposition 6, given by

$$d(f) = l(f) + \sum_{\substack{f' \in \mathcal{F}: l(f') < l(f) \text{ and} \\ l(f')n(f') > s(f')|\mathcal{J}^L(f')|}} s(f');$$

that is, both SGS and MGS employ Proposition 6, but the former uses $l(f)$, whereas the later uses $d(f)$. Second, MGS uses Inequality 3 in order to check whether the current

Algorithm 4 UNIQUE QUEUE STRATEGY (UQS)

```
while J ≠ ∅ do
    select j ∈ J with minimum release time
    ▷ Head moves to the right
    if h ≤ l(j) then
        ▷ While moves head to the right, service jobs
        while h ≤ l(j) do
            h := r(f(h)) + 1
            J := J \ J(f(h))
            update(J)
        ▷ Head moves to the left
    else
        h := l(f(j))
        update(J)
        h := r(f(h)) + 1
        J := J \ J(f(j))
        update(J)
```

Algorithm 5 SELECTIVE GREEDY STRATEGY (SGS)

```
while J ≠ ∅ do
    WritePhase(J)
    ▷ Stop if there are no more requests to the left
    while J^L(f(h)) > 0 do
        f := f(h)
        h := l(f)
        if n(f) ≠ 0 then                ▷ File f has pending job
            ▷ Condition of Proposition 6
            if l(f)n(f) > s(f)|J^L(f)| then
                MiniBatch({f})
                J := J \ J(f)
        update(J)
    Phase2(J)
```

file should be merged with its left neighbour in a single mini-batch. For the later feature, the algorithm maintains a queue Q, which contains all jobs that should be included on the next mini-batch.

The *hybrid greedy strategy* (HGS), described in Algorithm 7, results from a small modification of SGS, where $l(f)$ is substituted for $d(f)$ in the verification of the condition described by Proposition 6. Similarly, HGS is a variation of MGS which employs only atomic mini-batches, so HGS can be interpreted as a hybrid algorithm that combines ideas from SGS and MGS.

Computational experiments

We present in this section the results of the computational evaluation we performed on the algorithms described for the online LTSP. We were not able to identify previous work addressing the problem, except for the ones that simply sort requests based on their corresponding files' offsets on tapes, so we used UQS as the reference for the state of the art.

The dataset used as benchmark is composed of synthetic scenarios, each parametrized by the *number of files* $|\mathcal{F}|$ and a *time-horizon* H, which represents the number of time units

Algorithm 6 MERGING GREEDY STRATEGY (MGS)

```
while J ≠ ∅ do
    WritePhase(J)
    ▷ Stop if there are no more requests to the left
    while J^L(f(h)) > 0 do
        f := f(h)
        h := l(f)
        update(J)
        f' := f(h − 1)                  ▷ Left neighbour of f
        ▷ Check merge of mini-batches
        if n(f) ≠ 0 then
            ▷ Condition of Inequality 3
            z := (l(f) − l(f'))n(f)
            z := z + |J(f') ∩ J(f)|(l(f) − r(f'))
            if n(f') > 0 and s(f)n(f') > z then
                Q := Q ∪ {f}
                J := J \ J(f)
            else
                ▷ Modified condition of Proposition 6
                if d(f)n(f) > s(f)|J^L(f)| then
                    Q := Q ∪ {f}
                ▷ Execute mini-batch if |Q| > 0
                if |Q| > 0 then
                    MiniBatch(Q)
                    J := J \ J(f)
                    Q := ∅
    Phase2(J)
```

Algorithm 7 HYBRID GREEDY STRATEGY (HGS)

```
while J ≠ ∅ do
    WritePhase(J)
    ▷ Stop if there are no more requests to the left
    while J^L(f(h)) > 0 do
        f := f(h)
        h := l(f)
        if n(f) ≠ 0 then                ▷ File f has pending job
            ▷ Modified condition of Proposition 6
            if d(f)n(f) > s(f)|J^L(f)| then
                MiniBatch({f})
                J := J \ J(f)
        update(J)
    Phase2(J)
```

within which all requests are released. These scenarios were generated as follows.

In order to construct the sequence of requests associated with f in $\mathcal{F}$, we initially drawn an integer value λ_f uniformly from interval $[H/50, H/5]$; values for different files are independent and identically distributed. Afterwards, we drawn H/λ_f samples from the Poisson distribution parametrized with λ_f and generate the sequence of release times for requests associated with f by setting the arrival time of the i-th request as the sum of the first i values drawn from the distribution; the first request associated with f is a write request, whereas the others are read requests. Requests whose release times are larger than H are discarded. We as-

sume that the tape is completely empty in the beginning of the simulation. Finally, the size of each file (in "blocks") is drawn uniformly from interval $[1, 20]$; the velocity in which the head processes requests is equal to one block per time unit, that is, each file f is traversed in $s(f)$ time units.

The results of our computational experiments are presented in Table 1. The first column describes the parameters of the family of instances using the format $|\mathcal{F}|$-H, and the other columns show the sum of the overall response times produced by algorithms UQS, FGS, SGS, MGS, and HGS. In order to reduce bias, we generated 10 instances for each configuration $|\mathcal{F}|$-H, and each entry of the table contains the sum of all results. The overall response times entries are reported as the ratio between the results of the respective algorithm and the best result obtained by that family of instances; the result of the best algorithm for each instance is marked in bold and is always equal to **1.000**.

Overall, the experiments show a clear superiority of the strategies based on Proposition 6, with SGS being the winner among them. These results can be seen as the empirical counterparts of Propositions 3 and 4 for the online LTSP.

A key difference between MGS and SGS is the incorporation of merging operations in the former, through which a file is incorporated to a mini-batch whenever it satisfies the condition of Inequality 3 with its left neighbour. Note, though, that the condition of Proposition 6 is not checked for these files; moreover, the verification performed for the merging operations is local, i.e., involves only pairs of consecutive files, and not all pairs of files composing a mini-batch. For these reasons, MGS may choose sub-optimal mini-batches, and we believe that this explains the (relatively small) superiority of SGS over MGS. Moreover, HGS frequently delivered better results than MGS, which suggests that our strategy to construct non-atomic mini-batches was indeed not effective for this dataset.

UQS and FGS had relatively poor performances, with UQS having a considerably unstable behaviour. Namely, it delivered the best results in one scenario (beating the other strategies by more than 19%) and unacceptable results in others; for family 1000-100, for instance, the overall response times were more than 40 times larger than the others.

Finally, we observe that the implementation of Phase 2 had a significant impact on the results of FGS, SGS, MGS, and HGS. In preliminary versions of these algorithms, h was always moved to the end of the tape in Phase 2, but as write requests stop being generated relatively early in the simulations, this strategy was yielding losses that were being avoided by UQS. By setting the end of Phase 2 to the rightmost file with associated jobs, the performance of these algorithms increased considerably, making them clearly superior to UQS.

Instance	UQS	FGS	SGS	MGS	HGS
5-100	1.270	1.515	**1.000**	1.008	1.003
5-1000	1.033	1.197	**1.000**	1.094	1.052
5-10000	1.101	1.871	**1.000**	1.003	1.001
5-100000	1.264	2.076	**1.000**	**1.000**	**1.000**
25-100	2.597	1.200	1.009	**1.000**	**1.000**
25-1000	1.170	1.319	**1.000**	1.030	1.029
25-10000	1.131	1.088	**1.000**	1.011	1.038
25-100000	**1.000**	1.860	1.197	1.190	1.197
50-100	4.182	1.092	1.001	**1.000**	**1.000**
50-1000	1.401	1.796	**1.000**	1.021	1.020
50-10000	1.063	1.161	**1.000**	1.010	1.006
50-100000	1.065	1.101	**1.000**	1.021	1.015
100-100	4.808	1.047	**1.000**	**1.000**	**1.000**
100-1000	1.631	1.594	**1.000**	1.001	1.001
100-10000	1.153	1.285	1.002	**1.000**	1.002
100-100000	1.326	1.018	1.013	1.026	**1.000**
1000-100	43.933	1.003	**1.000**	**1.000**	**1.000**
1000-1000	5.127	1.036	1.001	**1.000**	**1.000**
1000-10000	1.693	1.540	**1.000**	1.002	1.002
1000-100000	1.263	1.276	**1.000**	1.011	1.011

Table 1: Computational results for the online LTSP

Conclusion

We investigated in this article the Linear Tape Scheduling Problem, which aims to identify strategies for the execution of read and write operations in single-tracked magnetic tapes that minimize the overall response times for read requests. LTSP is similar to classic combinatorial optimization problems, such as the Travelling Repairman Problem and the Dial-a-Ride Problem in a line, but peculiarities on the behaviour of magnetic tapes make LTSP an interesting problem on its own.

In practical settings, LTSP is an online problem, so we investigated both its offline and its online versions. From our analysis, we were able to identify structural properties of LTSP which helped us to design scheduling strategies that performed well on the synthetic dataset used as benchmark in this work. The complexity of the offline LTSP remains open, but the excellent performance of SGS, a relatively simple scheduling strategy, suggests that the "hard" scenarios might not occur so frequently in practice.

We also note that even with the simplifications we introduced to our model, it captures real-life arrangements of files on tapes. Yet, a natural next step would be to extend the model to capture the layout of multi-track serpentine tapes, where two consecutive tracks have different reading/writing directions, thus enabling our model to deal with a greater class of workloads. In future work, we also intend to further investigate the theoretical aspects of the problem (both online and offline) and to experiment the proposed algorithms and datasets with different behaviours.

References

Afrati, F.; Cosmadakis, S.; Papadimitriou, C. H.; Papageorgiou, G.; and Papakonstantinou, N. 1986. The complexity of the traveling repairman problem. *Informatique Théorique et Applications* 20(1):79–87.

Augustine, J. 2002. Offline and online variants of the traveling salesman problem. Master thesis, Dept. of Electrical and Computer Engineering, Louisiana State University.

Ausiello, G.; Feuerstein, E.; Leonardi, S.; Stougie, L.; and Talamo, M. 2001. Algorithms for the on-line travelling salesman. *Algorithmica* 29(4):560–581.

Bock, S. 2015. Solving the traveling repairman problem on a line

with general processing times and deadlines. *European Journal of Operational Research* 244(3):690–703.

Cavalli, A.; Dell'Agnello, L.; Ghiselli, A.; Gregori, D.; Magnoni, L.; Martelli, B.; Mazzucato, M.; Prosperini, A.; Ricci, P. P.; Ronchieri, E.; Sapunenko, V.; Vagnoni, V.; Vitlacil, D.; and Zappi, R. 2010. Storm-gpfs-tsm: A new approach to hierarchical storage management for the lhc experiments. *Journal of Physics: Conference Series* 219(7):072030.

Fontana, R. E.; Decad, G. M.; and Hetzler, S. R. 2013. The impact of areal density and millions of square inches (MSI) of produced memory on petabyte shipments of TAPE, NAND flash, and HDD storage class memories. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, 1–8.

Frank, J. C.; Miller, E. L.; Adams, I. F.; and Rosenthal, D. C. 2012. Evolutionary trends in a supercomputing tertiary storage environment. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, 411–419.

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co.

Hasan, R.; Tucek, J.; Stanton, P.; Yurcik, W.; Brumbaugh, L.; Rosendale, J.; and Boonstra, R. 2005. The techniques and challenges of immutable storage with applications in multimedia. In *Electronic Imaging 2005*, 41–52. International Society for Optics and Photonics.

Hillyer, B. K., and Silberschatz, A. 1996. On the modeling and performance characteristics of a serpentine tape drive. *SIGMETRICS Perform. Eval. Rev.* 24(1):170–179.

Karuno, Y., and Nagamochi, H. 2001. A polynomial time approximation scheme for the multi-vehicle scheduling problem on a path with release and handling times. In *Algorithms and Computation*, volume 2223 of *LNCS*, 36–48. Springer Berlin Heidelberg.

Karuno, Y., and Nagamochi, H. 2005. Scheduling vehicles on trees. *Pacific Journal of Optimization* 1(3):527–543.

Karuno, Y.; Nagamochi, H.; and Ibaraki, T. 1998. A 1.5-approximation for single- vehicle scheduling problem on a line with release and handling times. In *Japan-U.S.A. Symposium on Flexible Automation*, 1363–1366.

Murray, S.; Bahyl, V.; Cancio, G.; Cano, E.; Kotlyar, V.; Presti, G. L.; Re, G. L.; and Ponce, S. 2012. Tape write-efficiency improvements in CASTOR. *Journal of Physics: Conference Series* 396(4):042042.

Oracle. 2011. Redefining tape usage with storagetek tape tiering accelerator and storagetek in drive reclaim accelerator. Technical report, Oracle.

Paepe, W. D.; Lenstra, J.; Sgall, J.; Sitters, R.; and Stougie, L. 2004. Computer-aided complexity classification of dial-a-ride problems. *INFORMS Journal on Computing* 16(2):120–132.

Pease, D.; Amir, A.; Real, L. V.; Biskeborn, B.; Richmond, M.; and Abe, A. 2010. The Linear Tape File System. In Khatib, M. G.; He, X.; and Factor, M., eds., *MSST*, 1–8. IEEE Computer Society.

Prakash, V.; Zhao, X.; Wen, Y.; and Shi, W. 2013. Back to the future: Using magnetic tapes in cloud based storage infras- tructures. In Eyers, D. M., and Schwan, K., eds., *Middleware*, volume 8275 of *LNCS*, 328–347. Springer.

Psaraftis, H.; Solomon, M.; Magnanti, T.; and Kim, T.-U. 1990. Routing and scheduling on a shoreline with release times. *Management Science* 36(2):212–223.

Real, L. C. V.; Richmond, M.; Biskeborn, B.; and Pease, D. 2015. An I/O scheduler for dual-partitioned tapes. In *Networking, Architecture and Storage (NAS), 2015 IEEE International Conference on*, 234–243.

Sandstå, O., and Midtstraum, R. 1999. Low-cost access time model for serpentine tape drives. In *Mass Storage Systems, 1999. 16th IEEE Symposium on*, 116–127. IEEE.

Schaeffer, J., and Casanova, A. G. 2011. TReqS: The Tape REQuest Scheduler. *Journal of Physics: Conference Series* 331(4):042040.

Tsitsiklis, J. N. 1992. Special cases of traveling salesman and repairman problems with time windows. *Networks* 22(3):263–282.

Zhang, X.; Du, D.; Hughes, J.; and Kavuri, R. 2006. HPTFS: A High Performance Tape File System. In *Proceedings of 14th NASA Goddard/23rd IEEE conference on Mass Storage System and Technologies*.