Theoretical Computer Science 423 (2012) 11–18

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

A note on sorting buffers offline

Ho-Leung Chan [a], Nicole Megow [b], René Sitters [c,*], Rob van Stee [b]

[a] The University of Hong Kong, Hong Kong

[b] Max-Planck-Institut für Informatik, Saarbrücken, Germany

[c] Vrije Universiteit Amsterdam, The Netherlands

ARTICLE INFO

Article history:
Received 18 October 2010
Received in revised form 12 September 2011
Accepted 29 December 2011
Communicated by T. Erlebach

Keywords:
NP-hard
Approximation algorithm
Resource augmentation
Buffer sorting

ABSTRACT

We consider the offline sorting buffer problem. The input is a sequence of items of different types. All items must be processed one by one by a server. The server is equipped with a random-access buffer of limited capacity which can be used to rearrange items. The problem is to design a scheduling strategy that decides upon the order in which items from the buffer are sent to the server. Each type change incurs unit cost, and thus, the objective is to minimize the total number of type changes for serving the entire sequence. This problem is motivated by various applications in manufacturing processes and computer science, and it has attracted significant attention in the last few years. The main focus has been on online competitive algorithms. Surprisingly little is known on the basic offline problem.

In this paper, we show that the sorting buffer problem with uniform cost is NP-hard and, thus, close one of the most fundamental questions for the offline problem. On the positive side, we give an $O(1)$-approximation algorithm when the scheduler is given a buffer only slightly larger than double the original size. We also sketch a fast dynamic programming algorithm for the special case of buffer size 2.

1. Introduction

The *sorting buffer* problem results from the following scenario. The input is a sequence σ of n items of different types. We represent different types by different colors, i.e., each item i is associated with a color $c(i)$. The number of colors in the sequence is denoted by C. All items must be processed by a server. The server is equipped with a random-access buffer of limited capacity which can be used to rearrange the items. The items are moved one after another into the buffer that can hold at most k items. At any step, a scheduling algorithm chooses a color from the buffer, say c, and then all items in the buffer of color c are removed from the buffer and processed by the server. This creates space in the buffer, and the next items in the sequence will be moved to the buffer. If some of these new items have color c, they will be removed and processed immediately and it continues until no item in the buffer is of color c. The scheduling algorithm then chooses a new color and repeats, until all items in the sequence are removed for processing. The goal is to design a scheduling algorithm that minimizes the total number of color changes. The server has no color initially.

While the sorting buffer problem looks simple, it models a number of important problems in manufacturing processes, hardware design, computer graphics, file servers and information retrieval. For example, consider the sequencing problem in an automotive paint shop [12], where cars are painted in different colors. The cars traverse this production stage consecutively, and whenever a color change is necessary, this causes setup and cleaning cost. The goal is to minimize the

* Corresponding author. Tel.: +31 20 598 6016.
E-mail addresses: hlchan@cs.hku.hk (H.-L. Chan), nmegow@mpi-inf.mpg.de (N. Megow), rsitters@feweb.vu.nl, r.a.sitters@vu.nl (R. Sitters), vanstee@mpi-inf.mpg.de (R. van Stee).

0304-3975/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2011.12.077

total cost for changing colors. For an extended discussion on various applications and more references, we refer the readers to, e.g., [15,9,5].

1.1. Previous work

The sorting buffer problem (also known as *buffer reordering problem*) has attracted significant attention since it was first proposed by Räcke et al. [15]. The original focus was on competitive analysis of online algorithms. Räcke et al. [15] proposed an $O(\log^2 k)$-competitive algorithm and showed that some simple heuristics like First In First Out (FIFO) and Least Recently Used (LRU) are $\Omega(\sqrt{k})$-competitive, where $k > 0$ is the buffer size. Englert and Westermann [10] improved these results and gave an $O(\log k)$-competitive algorithm for a more general non-uniform cost function, where the cost of a color change depends on the final color. To obtain this result, they first relate their algorithm's solution to an optimal offline solution using a buffer of size $k/4$. Then, they prove that an offline optimum with buffer size $k/4$ is $O(\log k)$-competitive against an offline optimum with buffer size k. The first result translates into a constant competitiveness result using resource augmentation, i.e., their algorithm is 4-competitive when given a buffer with size 4 times the original size. The currently best known result was derived very recently by Adamaszek et al. [1]; they gave an $O(\sqrt{\log k})$-competitive deterministic online algorithm for the sorting buffer problem with non-uniform costs, as well as an almost matching lower bound of $\Omega(\sqrt{\log k/\log\log k})$.

Considerable work has been done for the problem when the cost function is a metric and the cost of a color change depends on both the original and final colors. We do not review the results here and refer the readers to the summary by Avigdor-Elgrabli and Rabani [5].

In order to develop good online methods, one of the most natural steps is to investigate the offline sorting buffer problem and identify its structural properties. Even if the offline problem is less relevant in practice, its analysis should be easier and give new insight to the problem. However, only little is known on the offline problem. It is easy to see that there are dynamic programming algorithms that solve the problem optimally in $O(n^{k+1})$ or $O(n^{C+1})$ time; see also [14,13]. Aiming at polynomial time algorithms, the above mentioned online algorithms already provide the best known offline approximation guarantees (which are non-constant). A constant approximate algorithm for the offline case on the line metric has been derived by Khandekar and Pandit [13]; however, it runs in quasi-polynomial time.

There has been research on the complementary variant of the sorting buffer problem, where the objective is to maximize the number of avoided color changes in the input stream. This problem is more successful in terms of approximation algorithms. Kohrt and Pruhs [14] gave a polynomial time 20-approximate algorithm. This was later improved by Bar-Yehuda and Laserson [6] who gave a 9-approximation algorithm for non-uniform cost. Note that the maximization and minimization problems have the same optimal solution, but they may be very different in terms of approximation.

1.2. Our results

We give a concise NP-hardness proof for the sorting buffer problem by a reduction from 3-Partition [11], and hence close one of the fundamental open questions on this problem [5]. Clearly, this implies that both variants, the minimization and the maximization problem, are NP-hard. Independently, an NP-hardness proof was given by Asahiro et al. [3]. However, their proof is much longer than ours and turned out to be incorrect [2]. Recently, they gave a new, though still very long, proof [4].

We also note, that increasing the number of servers does not make the problem easier. The idea of modeling more servers leads to an intuitive generalization of (or joint model for) the sorting buffer problem and the somewhat related well-known paging problem. In the latter problem, there is given a cache of m colors while a request from an online request sequence must be served immediately without intermediate buffering. We could interpret the cache as m servers that may immediately serve a current request. This leads to a *generalized sorting buffer problem* in which we have a buffer of size k and m servers. In this general formulation, the sorting buffer problem corresponds to the special case with $m = 1$, while the paging problem has $k = 1$. Yet, the earlier problem is NP-hard, as we show in this paper, while the later problem is polynomial-time solvable [7].

Naturally, we also consider an immediate adaption of the optimal paging algorithm [7] Longest Forward Distance (LFD) as a candidate for the sorting buffer problem. However, we show that it is $\Omega(k^{1/3})$-approximate, hence a different strategy is needed to derive constant approximate algorithms for sorting buffers. This negative result is in line with similar observations for several other natural (online) strategies in [15].

On the positive side, we consider the setting with resource augmentation, where the algorithm is given a larger buffer than the optimal one. We give a new LP formulation for the sorting buffer problem and show that it can be rounded using a larger buffer size. This gives an $O(1/\epsilon)$-approximate algorithm using a buffer of size $(2+\epsilon)$ times that of optimal.

We also sketch an $O(n \log C)$-time optimal algorithm for the special case in which the size of the buffer is $k = 2$. The algorithm uses a somewhat special dynamic programming approach with a non-trivial combination of data structures that guarantee the linear running time in the input size. Note that it is straightforward to obtain a dynamic programming algorithm that runs in $O(n^{k+1})$ time; our algorithm improves this significantly.

Organization. In Section 2, we show that the sorting buffer problem is NP-hard. We also generalize this proof to show that the sorting buffer problem with m servers is NP-hard for any constant $m \geq 1$. In Section 3, we provide the LP and the constant factor approximation algorithm using a larger buffer size, and finally we sketch in Section 4 the dynamic programming

algorithm for $k = 2$. We present the lower bound on the approximation ratio of LFD in Section 5. We conclude with some open problems in Section 6.

2. Complexity

An instance of the sorting buffer problem is given by a sequence of n items and for each item a number from $\{1, 2, \ldots, C\}$ which represents the color of the item. Hence, the size of the input of a sequence of n items is $\Omega(n \log C) = O(n \log n)$.

Theorem 1. *The sorting buffer problem is NP-hard.*

Proof. We reduce from 3-Partition which is known to be strongly NP-hard [11]: given $3q$ positive integers $a_1, a_2, \ldots, a_{3q}$ and an integer A such that $a_1 + a_2 + \cdots + a_{3q} = qA$, can we partition $\{1, 2, \ldots, 3q\}$ into q sets I_i such that $\sum_{j \in I_i} a_j = A$ for all $i \in \{1, 2, \ldots, q\}$?

Given an instance of 3-Partition, we construct an instance σ for the sorting buffer problem as follows. We multiply all the $3q$ integers by a large number $L = 2q^2A$. Let $b_j = La_j$ for all j and $B = LA$. We define the buffer size as $k = qB + L/2$. We see the buffer as having a *main* part of capacity qB and an *extra* part of capacity $L/2$. For each $j \in \{1, 2, \ldots, 3q\}$ we define a color j. We call these the *primary* colors. The sequence contains many other colors but we do not label those explicitly. We call those the *secondary* colors. The input sequence is defined by $3q + 4$ subsequences:

$$\sigma = \beta\gamma_1\delta_1\alpha_1\gamma_2\delta_2\alpha_2 \cdots \gamma_q\delta_q \ \alpha_q\gamma_{q+1}\delta_{q+1}\alpha_{q+1}.$$

As we will see, β and α_i are subsequences used to encode the numbers b_j and a_j. γ_i and δ_i are used as separators to enforce certain actions of the sorting buffer algorithm. The subsequences are defined as follows.

- β contains b_j items of color j for each $j \in \{1, 2, \ldots, 3q\}$. Items are given in arbitrary order. Note that the total number of items equals qB which is the size of the main part of the buffer.
- α_i $(i = 1 \ldots q + 1)$ contains a_j items of color j for each $j \in \{1, 2, \ldots, 3q\}$. Again, the order is arbitrary.
- γ_i $(i = 1 \ldots q + 1)$ we distinguish between $i \leq q$ and $i = q + 1$. For $i \leq q$ it starts with iB items of different colors followed by again one item of each of these colors. Any color used in γ_i is unique in the sense that it appears twice in γ_i and nowhere else in the sequence σ. Sequence γ_{q+1} is defined exactly the same but the number of colors is now $k - M$, where $M = \frac{1}{2}q(q + 1)A$. Hence γ_{q+1} contains $2(k - M)$ items. Note that $M \leq q^2A = L/2$.
- δ_i $(i = 1 \ldots q + 1)$ contains k items of the same color. This color is not used anywhere else in σ.

Let us, just for clarity, count the number of colors in σ. The subsequences β and α_i contain the $3q$ primary colors. The sequences δ_i together contain $q + 1$ colors. A sequence γ_i contains iB colors for $i \leq q$ and $k - M$ colors for $i = q + 1$ (each color twice). The total number of colors in the sequence σ is $C = 3q + q + 1 + (\sum_{i=1}^{q} iB) + k - M$. We shall prove that a 3-partition exists if and only if the sequence can be served with at most $C + 3q$ color switches.

First assume that a 3-Partition $I_1, \ldots, I_q$ exist. This is the easy direction and it immediately provides the reader with more insight in the reduction. Note that the buffer is large enough to hold all items from β plus all items from $\alpha_1, \alpha_2, \ldots, \alpha_q$ since the number of these is $qB + q^2A \leq k$. The server does the following. It places β in the buffer. Then it removes the B items with a color in I_1. It gives a space of B in the buffer to serve all items in γ_1 while using each color in γ_1 only once. Then the server switches to the color of δ_1 and removes all these items. Next all items of α_1 enter the buffer and the server removes all items with a color in I_2. Note that we have removed $2B$ items from β and A items from α_1 so far. This gives enough free space in the buffer to serve all items in γ_2 while using each color in γ_2 only once. Next the sequence δ_2 is served and α_2 enters the buffer. This process continues until α_q has entered the buffer. Let us count the number of items in the buffer at this moment. All primary colors have been used exactly once. Hence, all items from β are gone. All the qA items from α_q are in the buffer. From α_{q-1}, exactly $(q - 1)A$ items are still in the buffer. In general, iA items from α_i remain $(1 \leq i \leq q)$. Hence, the number of items in the buffer after α_q has entered the buffer is *exactly* $\sum_{i=1}^{q} iA = \frac{1}{2}q(q + 1)A = M$. (Later we shall argue that the number of remaining items is larger than M if no 3-partition exists.) We see that γ_{q+1} can be served using each of its colors only once. Next δ_{q+1} is served and, finally, α_{q+1} can be served by using each primary color once more. We conclude that σ can be served using each primary color twice and each secondary color once. This gives a total cost of $C + 3q$.

For the other direction, we assume that the optimal solution Opt can serve all items with $C + 3q$ color changes. We shall prove that a 3-Partition exists. We first list some properties that Opt has.

Claim 1. *Before the first item of α_i enters the buffer,* Opt *must have used the color of δ_i.*

Proof. The length of δ_i is equal to the buffer size. □

We remark that the reduction would be valid without the subsequences $\delta_1, \ldots, \delta_q$. However, these subsequences give us separations of the server sequence which enhance the analysis. Hence, we assume that the server switches to color δ_i simply once δ_i enters the buffer.

Claim 2. *We may assume that* Opt *serves γ_i completely before serving δ_i.*

Proof. Since the items of γ_i which remain in the buffer when Opt switches to color δ_i cannot be combined with items arriving later, we may as well serve these remaining items before switching to δ_i. □

It follows from the preceding two claims that we may assume that

Claim 3. OPT *serves the sequences* γ_i *and* δ_i *in the order* $\gamma_1\delta_1\gamma_2\delta_2\cdots\gamma_{q+1}\delta_{q+1}$.

For $1 \le i \le q$, let H_i be the set of primary colors used before serving δ_i. We want to show that a large number of items in β have been removed before serving δ_i.

Claim 4. *For all* $i \in \{1, 2, \ldots, q\}$, *we have* $\sum_{j\in H_i} b_j \ge iB$. *Hence,* $\sum_{j\in H_i} a_j \ge iA$.

Proof. Assume that $\sum_{j\in H_i} b_j < iB$ for some i. Then, $\sum_{j\in H_i} b_j \le iB - L$ since all b_j are multiples of L. This means that from the qB items of β at most $iB - L$ items are removed before time δ_i. Hence, the free space we have to serve γ_i is no more than $iB - L + L/2 \le iB - L/2$, where we add $L/2$ due to the extra part of the buffer. But then at least $L/2$ colors of γ_i must be used more than once. The total number of color switches will be at least $C + L/2 = C + q^2A > C + 3q$ for $qA > 3$.

The second statement follows by noticing that $b_j = La_j$ and $B = LA$. □

Claim 5. *Every primary color is used exactly two times: once before serving* δ_q *and once after serving* δ_{q+1}. *Every secondary color is used exactly once.*

Proof. Taking $i = q$ in Claim 4 we see that all $3q$ primary colors must be used before serving δ_q. Further, each primary color must also be chosen at least once after switching to δ_{q+1} since α_{q+1} contains all primary colors and is served after the switch to δ_{q+1}. We see that the bound of $C + 3q$ can only be reached if the statement in the claim holds. □

Let $I_1 = H_1$ and $I_i = H_i \setminus H_{i-1}$ for $2 \le i \le q$, i.e., the set of primary colors used between serving δ_{i-1} and serving δ_i. Consider the buffer contents just after serving δ_q. If $j \in I_i$ then the buffer contains at least $(q - i + 1)a_j$ items of color j. The total number of primary colored items in the buffer is at least

$$\sum_{i=1}^{q}\sum_{j\in I_i}(q-i+1)a_j = \sum_{i=1}^{q}\sum_{j\in H_i} a_j \ge \sum_{i=1}^{q} iA = \frac{1}{2}q(q+1)A = M. \tag{1}$$

If the inequality (1) is strict, then at least one color of γ_{q+1} is used twice which contradicts Claim 5. Hence, equality holds and this can only be true if equality in Claim 4 holds for all i. This implies that $\sum_{j\in I_i} a_j = A$ for all i. Hence, a 3-Partition exists. □

The NP-hardness of buffer sorting extends to the generalized sorting buffer problem with multiple servers m, even if m is constant.

Theorem 2. *The generalized sorting buffer problem is NP-hard for any number of servers* $m \ge 1$.

Proof. The case $m = 1$ is NP-hard, see Theorem 1. We show that the problem with m servers and buffer size k can be reduced to the problem with $m + 1$ servers and the same buffer size. The theorem then follows by induction.

Assume $m = \ell$ is NP-hard for some integer $\ell \ge 1$. Consider an arbitrary sequence ρ for the case $m = \ell$. We take a color x not used in ρ and add k items of color x between any two consecutive items in ρ. Let the resulting sequence be ρ'. We claim that it is optimal to ρ' to let one server serve only color x and the other ones the remaining colors. Suppose this were true, then ρ can be served using ℓ servers with minimum cost z if and only if ρ' can be served using $\ell + 1$ servers with minimum cost $z + 1$. Hence, the case for $m = \ell + 1$ is also NP-hard.

It is left to prove the claim. For the sake of contradiction, suppose there is an optimal solution OPT to sequence ρ' which does not serve all items of color x by the same server. Let m_1 be the server which serves the first item of color x in ρ'. Consider the first moment in which an item preempts the sequence of consecutively serving color x by m_1, i.e., an item i of color $c(i) \ne x$ is assigned to m_1. Let S be the set of items that are in the buffer at that moment. We can assume that the next item j that enters the buffer is the first of k consecutively incoming color-x items. (Whenever we remove one color-x item from the buffer, then we can serve all of them without extra cost.) Hence, with the buffer capacity k, OPT must serve at least one (and thus w.l.o.g. all) items of color x before an new item with color different from x can enter the buffer.

Consider the schedule after OPT served the color x items by some server, say m_2. Suppose $m_1 \ne m_2$. While the current color of m_2 is x, server m_1 might have served after i some items of the same or other colors from S; let i' be the last item assigned to m_1 so far. Now, we simply exchange the current output sequence on server m_1 from item i up to i', with the sequence of color-x items on m_2. This is feasible since we only swap output positions of items in S that are in the buffer or enter with the same color x. Note, that the currently active colors of the servers are not changed. Moreover, the cost of the schedule can only decrease: moving the color-x items to m_1 reduces the cost by one and moving the sequence starting with item i to m_2 does not cause a new color change. Thus, OPT was not an optimal solution.

If $m_1 = m_2$, then we extract from the output sequence on m_1 the subsequence i up to i', and assign it to the end of the current sequence of some server, say m_2. Clearly, the current color of m_2 changes and may cause an additional unit of cost when OPT assigns the next item to m_2. However, we reduce the cost by one unit when removing the color change on m_1 for switching back to color x. Thus, the cost does not increase. This exchange can be applied iteratively to an optimal solution until no items of a color different from x is assigned to m_1. □

3. Resource augmentation

In this section, we give an LP-based algorithm which yields an $O(1/\epsilon)$-approximation with respect to the optimal solution that uses no more than $1/2 - 2\epsilon$ times the original buffer size. By scaling up the buffer size by a factor of $2 + O(\epsilon)$, it gives an $O(1/\epsilon)$-approximate algorithm using a buffer size of $2 + \epsilon$ times that of optimal.

We first introduce a new LP relaxation, followed by a rounding scheme. We consider that the buffer is empty initially. For each time step $i = 1, 2, \ldots, n$, the following three events occur. (1) The i-th item is moved to the buffer, (2) the algorithm chooses $c(i)$ to be the color of the buffer, and (3) all items in the buffer with color $c(i)$ are removed. Call an interval a c-interval if the color of the buffer is c throughout the interval and call it *non-c* if the color is not c throughout the interval. The cost for serving a color c is the number of maximal c-intervals. Note that the cost over all colors is exactly $2 - C$ plus the number of maximal non-c intervals for each color c. One observation is that after each time step $i = 1, 2, \ldots, n$, the number of items in the buffer should be at most $k - 1$. It motivates the following IP.

We define a variable $y^c_{s,t}$ for every color c and time steps s, t with $1 \le s \le t \le n$. $y^c_{s,t}$ should be one if $[s, t]$ is a maximal non-c interval; and it is zero otherwise. For each color c and time step $s \le i$, let $A^c_{s,i}$ be the number of items with color c moved into the buffer during $[s, i]$.

$$\text{minimize } 2 - C + \sum_c \sum_{s,t:\ s\le t} y^c_{s,t}$$

$$\text{subject to } \sum_{s,t:\ s\le t;\ s\le i+1;\ i\le t} y^c_{s,t} \le 1 \quad \text{for all } c \text{ and } i = 1, 2, \ldots, n+k-1 \tag{2}$$

$$\sum_c \sum_{s,t:\ s\le i\le t} y^c_{s,t} = C - 1 \quad \text{for all } i = 1, 2, \ldots, n+k-1 \tag{3}$$

$$\sum_c \sum_{s,t:\ s\le i\le t} A^c_{s,i} y^c_{s,t} \le k - 1 \quad \text{for all } i = 1, 2, \ldots, n-1 \tag{4}$$

$$\sum_{s:\ s\le i} A^c_{s,i} y^c_{s,i} = 0 \quad \text{for all } c \text{ and } i = n+k-1 \tag{5}$$

$$y^c_{s,t} \in \{0, 1\} \quad \text{for all } c \text{ and } s, t \in \{1, 2, \ldots, n+k-1\}. \tag{6}$$

The first constraint (2) ensures two things: (i) for any color c and time i, i is included in at most one maximal non-c interval and (ii) maximal non-c intervals are really maximal, i.e., if $y^c_{s,t} = y^c_{u,v} = 1$ then $t \le u + 2$ or $v \le s + 2$. By (i), each color c contributes at most 1 to the left hand side of the second constraint (3). Hence this constraint ensures that at any time i, the color of the buffer is different from exactly $C - 1$ colors. Constraint (4) ensures that by the end of each time step $i \le n - 1$, the number of items remaining in the buffer is at most $k - 1$ and constraint (5) ensures that the buffer is empty at the end. It is easy to verify that for any valid schedule we can set the values of $y^c_{s,t}$ according to whether it is a maximal non-c interval and this satisfies all the constraints. On the other hand, any IP-solution corresponds with a feasible coloring sequence with the same cost. The LP-relaxation is obtained by replacing (6) with $y^c_{s,t} \ge 0$. It is easy to verify that any LP-solution has value at least C. We can round the LP to get an $O(1/\epsilon)$-approximation against an optimal solution that uses no more than $1/2 - 2\epsilon$ times the buffer size. Define

$$x^c_i = \sum_{\substack{s,t:\\ s\le i\le t}} y^c_{s,t}.$$

Intuitively, $1 - x^c_i$ is the fraction of color c on the machine at step i. Further, define

$$z^c_i = \sum_{\substack{s:\\ 1\le s\le i}} y^c_{s,i}, \quad \text{and} \quad Z^c_i = \sum_{j=1}^{i} z^c_j.$$

The variable z^c_i sums over all intervals ending in i and the variable Z^c_i sums over all intervals ending in i or before that. In particular, Z^c_n is the LP-cost for color c. The value Z^c_i is non-decreasing in i. We mark every step that Z^c_i increases by another ϵ. More precisely, mark the first step i for which $Z^c_i \ge \epsilon$ and mark every next step i' for which $Z^c_{i'}$ has increased by at least ϵ since the last marking.

A feasible integral solution is found by the following rounding scheme.

LP Rounding. Start with an arbitrary buffer color. For $i = 1$ to $n + k - 1$ do:

(i) Remove all items with the current color (state) of the buffer.
(ii) For each marked color c, remove all its items.
(iii) If $x^{c'}_i \le 1/2 - \epsilon$ for some c', then switch the color to c' and remove all items with color c'.

Theorem 3. *The LP Rounding Algorithm applied to an optimal LP solution yields an $O(1/\epsilon)$-approximate solution for the sorting buffer problem when the optimum is using a buffer of size at most $1/2 - 2\epsilon$ times the original buffer size k.*

Proof. First we argue that (iii) is well defined. Constraint (3) states that $\sum_c x_i^c \geq C - 1$ and (2) states $x_i^c \leq 1$. Hence, there is at most one c' for which $x_i^{c'} \leq 1/2 - \epsilon$. $(*)$

The first step (i) is done for free, and one can easily verify that only the just entered item is possibly removed in this step. Clearly, the number of markings is $O(1/\epsilon)$ times the LP cost. Consider two consecutive switches. If at least one of the two is due to a marking then we charge both to the marking. To prove that the total number of switches is $O(1/\epsilon)$ times the LP cost we only need to bound the number of pairs of consecutive switches in which both are of type (iii). Assume the buffer switches to c' in step i and subsequently switches to another color c'' in step $j > i$ and both are of type (iii). We have $x_i^{c'} \leq 1/2 - \epsilon$ and $x_j^{c''} \leq 1/2 - \epsilon$. The first implies that $x_i^{c''} \geq 1/2 + \epsilon$; see $(*)$. Hence, $x_j^{c''} - x_i^{c''} \leq -2\epsilon$.

Notice that for every $j > i$ and c holds that

$$\begin{aligned} x_j^c - x_i^c &= \sum_{\substack{s,t:\\ s\leq j\leq t}} y_{s,t}^c - \sum_{\substack{s,t:\\ s\leq i\leq t}} y_{s,t}^c = \sum_{\substack{s,t:\\ i+1\leq s\leq j\leq t}} y_{s,t}^c - \sum_{\substack{s,t:\\ s\leq i\leq t\leq j-1}} y_{s,t}^c \geq 0 - \sum_{\substack{s,t:\\ s\leq i\leq t\leq j-1}} y_{s,t}^c \\ &= -(Z_{j-1}^c - Z_{i-1}^c). \end{aligned}$$

Therefore, $2\epsilon \leq x_i^{c''} - x_j^{c''} \leq Z_{j-1}^{c''} - Z_{i-1}^{c''}$. Thus, for color c'' there is an increase of the Z-variable of 2ϵ between two switches of the third type. We conclude that the total cost due to switches of the third type is also $O(1/\epsilon)$ times the LP cost.

Now we bound the capacity needed. Consider any c and step j and let $i < j$ be the last time before j that c was removed from the buffer in the rounded solution. We may assume that c was not removed at step j since otherwise there are no items of color c at the end of step j. Denote the term for color c in constraint (4) by a_j^c.

$$a_j^c = \sum_{\substack{s,t:\\ s\leq j\leq t}} A_{s,j}^c y_{s,t}^c.$$

Intuitively, a_j^c is the amount of color c in the buffer at step j in the LP-solution. On the other hand, the number of items of color c in the buffer at step j in the rounded solution is $A_{i+1,j}^c$. To relate the rounded solution to the LP-solution we are interested in the variables that correspond to (s,t)-intervals with $s \leq i+1 \leq j \leq t$. For these (s,t)-intervals we have $A_{s,t}^c \geq A_{i+1,j}^c$.

Since we have not picked color c in steps $i+1,\ldots,j$, we have $Z_j^c - Z_i^c < \epsilon$. Note further that $\sum_{s,t:s\leq i+1\leq j\leq t} y_{s,t}^c \geq x_{i+1}^c - Z_{j-1}^c + Z_i^c$. Since c is not removed at step $i+1$ we have $x_{i+1}^c > 1/2 - \epsilon$. Using additionally $Z_{j-1}^c \leq Z_j^c$ we conclude that

$$\sum_{\substack{s,t:\\ s\leq i+1\leq j\leq t}} y_{s,t}^c > \frac{1}{2} - \epsilon - Z_{j-1}^c + Z_i^c \geq \frac{1}{2} - \epsilon - Z_j^c + Z_i^c > \frac{1}{2} - 2\epsilon.$$

Finally, we can relate the amount of c in the LP-buffer with the number of c in the buffer of the rounded solution.

$$a_j^c = \sum_{\substack{s,t:\\ s\leq j\leq t}} A_{s,j}^c y_{s,t}^c \geq \sum_{\substack{s,t:\\ s\leq i+1\leq j\leq t}} A_{s,j}^c y_{s,t}^c \geq A_{i+1,j}^c \sum_{\substack{s,t:\\ s\leq i+1\leq j\leq t}} y_{s,t}^c \geq A_{i+1,j}^c \left(\frac{1}{2} - 2\epsilon\right).$$

Hence, the total number of items in the buffer after step j is $\sum_c A_{i+1,j}^c \leq \sum_c a_j^c/(1/2-2\epsilon) \leq (k-1)/(1/2-2\epsilon)$. Moreover, when $j = n+k-1$, we have $a_j^C = 0$ by constraint (5). This implies that the buffer is empty at the end. □

4. Dynamic programming

Straightforward dynamic programming algorithms solve the sorting buffer problem optimally in running time $O(n^{k+1})$ or $O(n^{C+1})$; see also [14,13]. In this section we consider the special problem setting with a buffer of size $k = 2$. We have the following Theorem.

Theorem 4. *There is an optimal algorithm solving the sorting buffer problem with buffer size $k = 2$ in time $O(n \log C)$.*

Note that the size of the input is $O(n \log C)$, so this running time is optimal. In our dynamic programming algorithm, we maintain the optimal cost $\textsc{Opt}_i$, a set S_i of colors, and the sizes of those colors. A color c is in S_i if there exists an optimal way to serve the first i items in the sequence such that an item of color c is served last. The size of a color is the (or, a possible) number of items of this color that are served together if this color is served last. In order to use only linear time, from one step to the next we only store the *changes* in S_i and in the sizes of the colors. This works because the number of these changes is an amortized constant per step.

Observation 1. *For each $i > 1$, we have $|S_i| \leq |S_{i-1}| + 1$.*

The only color that could possibly enter the set of optimal finishing colors is the color of the most recent item; any other color would have been optimal before. The following lemma is crucial.

Lemma 5. *At any step i, there can be at most one color c such that $size(c) > 1$; this is color c_i.*

Proof. Suppose there is any other color c in S_i with $size(c) > 1$. Then the last two items in some optimal serving order have color $c \neq c_i$. But then item i is served in step $i - 2$ or before, i.e., before it entered the buffer, a contradiction. □

The main technical difficulty to get a linear running time was storing and retrieving the necessary data efficiently, so that the program does not just calculate the optimal cost but also returns an optimal schedule in linear time. For the dynamic program, its implementation and analysis, see [8].

5. A lower bound for LFD

The well-known paging problem has several offline algorithms that solve it to optimality. One of those is the *Longest Forward Distance* (LFD) algorithm [7]. With the mentioned relation to the sorting buffer problem, it is reasonable to consider a natural adaption of this algorithm for sorting buffer. In the following we give a negative result that rules out LFD as a candidate for a constant approximation algorithm.

Longest Forward Distance (LFD).
If no item can be served without a color change, then choose the color of item i that has its next occurrence $j > i$ farthest in the future of the sequence. If no more items j with the same color as i exist, the distance is infinity.

Theorem 6. *LFD has an approximation ratio of at least $\Omega(k^{1/3})$.*

Proof. Consider the following input instance. Given is a buffer of size $M + n$, where $M \geq n^3$. The sequence of items is as follows; we describe each item by its color (natural number), and we denote by a^b that the item with color a appears b times consecutively.

$[\ 0^M\ ]$
$[\ 123\ldots n\ ]\ [\ 2\ 3^2\ 4^3\ \ldots\ n^{n-1}\ ]$
$[\ 0123\ldots n-1\ ]\ [\ 2\ 3^2\ 4^3\ \ldots\ (n-1)^{n-2}\ ]$
$[\ 0123\ldots n-2\ ]\ [\ 2\ 3^2\ 4^3\ \ldots\ (n-2)^{n-3}\ ]$
$\ldots$
$[\ 0123\ ]\ [\ 2\ 3^2\ ]$
$[\ 012\ ]\ [\ 2\ ]$
$[\ 01\ ].$

The sequence consists of $n + 1$ lines; let us denote them as $L_0, L_2, \ldots, L_n$. Initially, the buffer contains all items of line L_0 and the first block (in brackets) of L_1. An optimal solution chooses color 0 first; it can serve all items of this color and by the end, all remaining items of the sequence are in the buffer. Thus, there are no more than $n + 1$ color changes necessary.

LFD chooses color 1 first, moving the next item of color 2 into the buffer. Then it picks 2, moving two items of color 3 into the buffer and repeats until it chooses n and moves the first block of L_2 into the buffer. Then the process repeats. This way, LFD causes $n - i$ color changes serving the first block of line L_i. Thus, it has total cost $n(n + 1)/2$.

The ratio of LFD's cost and the optimal cost for this sequence are $n/2$. Hence, LFD has an approximation ratio bounded by $\Omega(k^{1/3})$ for a given buffer of size k. □

6. Open problems

Now that NP-hardness has been settled, the main open problem is to design a polynomial time constant factor approximation. In the introduction we listed several partial results on this. Given our LP-rounding result, a natural next step is to design an algorithm that gives an $O(1/\epsilon)$-approximation against an offline solution using only $(1 - \epsilon)k$ capacity, instead of $(1/2 - 2\epsilon)k$.

We gave a dynamic program for $k = 2$ which has a significantly better running time than the straightforward DP. It would be interesting to give an exact algorithm with a running time that is much less than $O(n^{k+1})$.

Our NP-completeness proof does not answer the question of how well this problem can be approximated in polynomial time. Hence it remains open whether the buffer sorting problem is APX-hard or not.

Acknowledgement

The fourth author's research was supported by the German Research Foundation (DFG) under grant no. STE 1727/3-2.

References

[1] A. Adamaszek, A. Czumaj, M. Englert, H. Räcke, Almost tight bounds for reordering buffer management, in: Proceedings of the 43rd ACM Symposium on Theory of Computing, 2011, pp. 607–616.
[2] Y. Asahiro, Private communication, 2010.
[3] Y. Asahiro, K. Kawahara, E. Miyano, NP-hardness of the sorting buffer problem on the uniform metric, in: Proceedings of the 2008 International Conference on Foundations of Computer Science, CSREA Press, 2008, pp. 137–143.
[4] Y. Asahiro, K. Kawahara, E. Miyano, NP-hardness of the sorting buffer problem on the uniform metric (2010).
[5] N. Avigdor-Elgrabli, Y. Rabani, An improved competitive algorithm for reordering buffer management, in: M. Charikar (Ed.), Proc. of the 21st SODA, 2010, pp. 13–21.
[6] R. Bar-Yehuda, J. Laserson, Exploiting locality: approximating sorting buffers, Journal on Discrete Algorithms 5 (4) (2007) 729–738.
[7] L. Belady, A study of replacement algorithms for virtual storage computers, IBM Systems Journal 5 (1966) 78–101.
[8] H.-L. Chan, N. Megow, R. van Stee, R. Sitters, The sorting buffer problem is NP-hard. CoRR, abs/1009.4355, 2010.
[9] M. Englert, H. Räcke, M. Westermann, Reordering buffers for general metric spaces, in: Proc. of 39th STOC, 2007, pp. 556–564.
[10] M. Englert, M. Westermann, Reordering buffer management for non-uniform cost models, in: Proc. of 32th ICALP, 2005, pp. 627–638.
[11] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman and Company, New York, 1979.
[12] K. Gutenschwager, S. Spieckermann, S. Voß, A sequential ordering problem in automotive paint shops, International Journal of Production Research 42 (9) (2004) 1865–1878.
[13] R. Khandekar, V. Pandit, Online and offline algorithms for the sorting buffers problem on the line metric, Journal of Discrete Algorithms 8 (1) (2010) 24–35.
[14] J.S. Kohrt, K. Pruhs, A constant approximation algorithm for sorting buffers, in: Proc. of LATIN, 2004, pp. 193–202.
[15] H. Räcke, C. Sohler, M. Westermann, Online scheduling for sorting buffers, in: Proc. of 10th ESA, 2002, pp. 820–832.