Reordering Buffer Management with Advice

Anna Adamaszek[1⋆], Marc P. Renault[2], Adi Rosén[3], and Rob van Stee[4⋆⋆]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
anna@mpi-inf.mpg.de
[2] Université Paris Diderot - Paris 7 and UPMC, France
mrenault@liafa.univ-paris-diderot.fr
[3] CNRS and Université Paris Diderot - Paris 7, France
adiro@liafa.univ-paris-diderot.fr
[4] University of Leicester, Leicester, United Kingdom
rob.vanstee@le.ac.uk

Abstract. In the reordering buffer management problem, a sequence of colored items arrives at a service station to be processed. Each color change between two consecutively processed items generates cost. A reordering buffer of capacity k items can be used to preprocess the input sequence in order to decrease the number of color changes. The goal is to find a scheduling strategy that, using the reordering buffer, minimizes the number of color changes in the given sequence of items.
We consider the problem in the setting of online computation with advice. In this model, the color of an item becomes known only at the time when the item enters the reordering buffer. Additionally, together with each item entering the buffer, we get a fixed number of advice bits, which can be seen as information about the future or as information about an optimal solution (or an approximation thereof) for the whole input sequence. We show that for any $\varepsilon > 0$ there is a $(1+\varepsilon)$-competitive algorithm for the problem which uses only a constant (depending on ε) number of advice bits per input item.
We complement the above result by presenting a lower bound of $\Omega(\log k)$ bits of advice per request for an optimal deterministic algorithm.

1 Introduction

In the *reordering buffer management problem*, a sequence of colored items arrives at a service station for processing. At any time, the service station is configured to process items of a certain color c. Changing the configuration of the service station, i.e., preparing the station to service items of a different color, generates cost. The total cost of processing a sequence of items is equal to the number of color changes in the sequence. In order to reduce the processing cost, the service station is equipped with a *reordering buffer* that has capacity to hold k items. At each point in time, the buffer contains the first k items of the input sequence

⋆ Supported by the Alexander von Humboldt Foundation.
⋆⋆ Work performed while the author was at the Max-Planck-Institut für Informatik, Saarbrücken, Germany.

that have not yet been processed, and one arbitrary item has to be chosen to be extracted from the buffer and processed by the service station. If the color of the chosen item differs from the color of the previously processed item, the configuration of the service station has to be changed, generating cost. The goal is to find a scheduling strategy that, using the reordering buffer, preprocesses the input sequence to minimize the number of color changes. In the *online setting*, which is considered in this paper, the color of an item becomes known only at the time when the item enters the reordering buffer, i.e., we do not know the whole input sequence in advance.

This framework has many applications in various areas, such as production engineering, storage systems, network optimization and computer graphics (see [3, 6, 14, 17, 18] for more details). One simple example is a paint shop of a car manufacturing plant, where switching colors between two consecutively painted cars generates costs and delays due to the necessary cleaning and set-up. Therefore, some paint shops are equipped with a reordering buffer to preprocess the sequence of incoming cars (see [14]).

In this paper, we study the problem in the setting of *online computation with advice* [11]. This setting has received much attention in recent years as it relaxes the traditional online setting of no information about the future, and allows the online algorithm to get some partial information about future requests or about the structure of an optimal (or near-optimal) solution. It also allows one to quantify the amount of information about the future available to the online algorithm and to study the interplay between the amount of information and the attainable competitive ratio. Informally (See Section 2 for a formal definition), in this setting, the online algorithm receives with each request some b bits of *advice* which are the value of a function, defined by the algorithm, of the whole input sequence (including the future). In this manner the online algorithm receives some information about the future. Note that we view the advice as given by an oracle, i.e., we do not consider the issue if the advice can be computed efficiently when knowing the whole input sequence.

Related work. The reordering buffer management problem has been introduced by Räcke et al. [18], and has been extensively studied. In the online setting, the best known results are a deterministic $O(\sqrt{\log k})$-competitive algorithm by Adamaszek et al. [1], and a randomized $O(\log \log k)$-competitive algorithm by Avigdor-Elgrabli and Rabani [5]. To complement this, there are (nearly) matching lower bounds of $\Omega(\sqrt{\log k / \log \log k})$ and $\Omega(\log \log k)$ on the competitive ratio of any online deterministic and randomized algorithms, respectively [1]. In the offline setting, i.e., when the whole input sequence is known in advance, the best known result is a constant factor approximation by Avigdor-Elgrabli and Rabani [4], while the problem is known to be NP-hard [9, 2].

More general versions of the problem have been studied, where the context switching cost for switching from an item of color c to an item of color c' depends on c' (e.g., [13, 1]), or on both c and c' [12].

The model of online computation with advice considered in the present paper has been introduced by Emek et al. [11]. In that paper, the authors give tight

bounds of $\Theta(\log n/b)$ on the competitive ratio of deterministic and randomized online algorithms with advice for metrical task systems, where n is the number of states of the system and b is the number of advice bits per request. They also give a deterministic online algorithm with advice for the k-server problem which is $k^{O(\frac{1}{b})}$-competitive, where $\Theta(1) \leq b < \log k$. This has been first improved by Böckenhauer et al. [7], and subsequently by Renault and Rosén [19] to $\left\lceil \frac{\lceil \log k \rceil}{b-2} \right\rceil$.

Böckenhauer et al. [8] introduced a somewhat similar model for online algorithms with advice, where the advice is a single tape of bits instead of being given separately for each request. This allows an algorithm to read a different number of bits of advice per request, but it also requires that the online algorithm knows how many bits of advice to read with each request. Thus, the two models are, in general, incomparable. Several results have been given in this related model [10, 8, 15, 16, 7]. For example, in [10, 8], the authors explore the number of bits of advice required for deterministic and randomized paging algorithms, algorithms for the DiffServ problem, algorithms for a special case of the job shop scheduling problem, and algorithms for the disjoint path allocation problem, to be 1-competitive.

Our contribution. We give an online algorithm with advice for the reordering buffer management problem that, for any $\varepsilon > 0$, achieves a competitive ratio of $1 + \varepsilon$, using only $O(\log(1/\varepsilon))$ advice bits per request. For any input sequence, we show how to construct the advice, based on an optimal solution for the given sequence, which allows us to obtain a good competitive ratio. The overview of the construction is as follows. The advice bits for each element of a color c encode how the algorithm should handle all "adjacent" elements of color c, i.e., if the algorithm should keep all elements of color c in the buffer until more items of color c arrive, output them at once, or output them, but only after a certain waiting period. The idea is that the order of the colors with the latter property is not contained in the advice of single elements (as that would require too many bits of advice), but it is encoded among all advice bits of the elements of the given color. The key obstacle is that with a small number of advice bits per item we cannot encode the exact order in which the colors should be output, in particular when a small number of elements of some color has to wait for a long time in the buffer. To deal with this problem, we modify the optimal solution by selecting some elements which will be removed earlier from the buffer. That frees additional space in the buffer, which allows us to keep some other elements longer in the buffer, until one more item of their color arrives, after which we can output these elements immediately. That significantly shortens the list of colors which have to be output "soon, but not yet". After this operation, we can encode the desired position in the list for all but a small fraction of elements (which also must be removed earlier from the buffer). We upper bound the increase in the cost of the generated output sequence by charging the additional color changes to the color changes of the optimal solution.

We complement the above result by showing that in order for an online algorithm to have optimal performance, the number of bits of advice must depend

on k. More precisely, $\Omega(\log k)$ bits are required. This lower bound applies even if all the advice bits are given to the algorithm before the sequence starts, and it matches (up to a constant factor) the trivial upper bound, where the advice indicates which color switch has to be performed at each step.

2 The Model

We use the definition of *deterministic online algorithms with advice* as presented in [11]. An online algorithm is defined as a request-answer game that consists of a request set R, a sequence of finite nonempty answer sets $A_1, A_2, \ldots$, and a sequence of cost functions $\text{cost}_n : R^n \times A_1 \times A_2 \times \cdots \times A_n \to \mathbb{R}^+ \cup \{\infty\}$ for $n = 1, 2, \ldots$. In addition, there is an advice space U of size 2^b, where $b \geq 0$ is the number of bits of advice provided to the algorithm with each request. With each request, the online algorithm receives some advice that is defined by a function, $u_i : R^* \to U$, where i is the request index, that is applied to the whole request sequence R^*, including future requests. A deterministic online algorithm with advice can, thus, be represented as a sequence of pairs (g_i, u_i), where g_i is the function defining the action of the online algorithm at step i and is defined $g_i : R^i \times U^i \to A_i$ for $i = 1, 2, \ldots$. The action taken by the online algorithm after receiving request r_i is therefore a function of the first i requests, $r_1, \ldots, r_i$, and the advice received so far, $u_1(\sigma), \ldots, u_i(\sigma)$.

We use the standard definitions of competitive analysis. We say that an algorithm is *α-competitive* if, for every finite request sequence σ, $|\text{ALG}(\sigma)| \leq \alpha \cdot |\text{OPT}(\sigma)| + \zeta$, where ζ is a constant which does not depend on the request sequence σ, $|\text{ALG}(\sigma)|$ denotes the cost of the solution generated by ALG for σ, and $|\text{OPT}(\sigma)|$ is the cost of an optimal solution for σ.

3 Structure of Advice and the Algorithm

The advice for each input element e consists of two parts: the *type* t_e and the *value* v_e. The type of an element e describes how e, or rather a whole collection of elements of the same color as e, should be handled by the algorithm ALG, i.e., if ALG should keep them in the buffer until more elements of the same color arrive, if ALG should output them at once, or output them, but only after a certain waiting period. There are four possible types of an element, i.e., $t_e \in \{\text{WAIT}, \text{LIST}, \text{READY}, \text{COMPLETE}\}$. The value v_e of an element e is used to encode an order, according to which some input elements will be output. For each element e we have $v_e \in \{0, \ldots, D-1\}$, where D is a constant depending on ε and will be fixed later.

The advice sequence will be constructed based on an optimal solution OPT for the input sequence.

Time. In this paper, we use the following notion of time. In each time step $1, \ldots, n$, one element arrives and is stored in the buffer. In each time step $k, \ldots, k+n-1$, one element is removed from the buffer, after the element which arrives at that time (if any) has been stored.

Color blocks. The elements which have the same color are partitioned into *color blocks*. A *complete color block* is a maximal set of elements of one color ending with an element of type COMPLETE, and containing no other elements of type COMPLETE. The advice sequence will be constructed in such a way that the types of the consecutive elements from one complete color block always form an expression

$$(\text{WAIT})^* \left((\text{LIST})^* \cup (\text{READY})^*\right) \text{COMPLETE},$$

i.e., first there is an arbitrary number (possibly zero) of elements of type WAIT, then an arbitrary number of elements of type LIST or an arbitrary number of type READY, and at the end exactly one element of type COMPLETE.

At a time t, a *color block* is that part of a complete color block which is contained in the buffer (some elements of the complete color block may have been already served, and some may not have been read yet). The notion of a color block here differs from the standard definition of a color block for the reordering buffer management problem, where it denoted the set of all elements of one color contained in the buffer, which were then output consecutively from the buffer. We now can have more than one color block of the same color in the buffer. The advice sequence will be constructed in such a way, that the algorithm ALG will always output all elements of each complete color block consecutively.

We define the *type t_B of a color block B* in the following way. If there is an element of type COMPLETE in B, then the type of B is COMPLETE. If all the elements of B have type WAIT, B has type WAIT. Otherwise, the type of B is LIST or READY, depending on whether B contains elements of type LIST or of type READY.

The values v_e of all elements of a color block B together encode a *value of a color block* $v_B \in \{0, \ldots, D^{|B|} - 1\}$. The value of a color block will be encoded only for color blocks containing elements of type LIST, and it will define the order in which these color blocks will be output by the algorithm ALG.

Waiting list. The algorithm ALG maintains a *waiting list* of color blocks of type LIST. This list contains only (a subset of) those color blocks that have the property that some elements of their color are read into the buffer of OPT while OPT is serving it. At the beginning of the algorithm, the waiting list is empty. Whenever a color block B of type LIST appears (i.e. the first element of type LIST for some color block is read from the input), it is inserted into the waiting list. The *initial* position of the block on the waiting list, i.e., the position where the block is inserted, is defined by the value v_B of the block. Value 0 denotes the head of the list. Note that when a new color block is inserted into the waiting list, the position of other blocks on the list can change.

The value v_B of each color block is read from the advice of its elements. However, some short color blocks may not contain sufficiently many elements to encode their required position v_B. We will check for this case and ensure that such blocks are never stored in the waiting list. Instead, we serve parts of them immediately and keep only part in the buffer.

The algorithm. Starting at time k, the algorithm ALG chooses a color block to be output in the following way.

1. If ALG has a complete color block in the buffer, it chooses the oldest of them (i.e., the block which first became complete is chosen).
2. Otherwise, if ALG has a color block of type READY in the buffer, it chooses the oldest of them (i.e., the block which first obtained type READY is chosen).
3. Otherwise ALG chooses a color block of type LIST which is at the head of the waiting list. The color block is then removed from the waiting list.

The algorithm then outputs consecutively all elements of the chosen color block, one element at each time step. Notice that when ALG starts outputting a color block, possibly some elements of the block have not yet been read from the input. When ALG gets such elements from the input sequence, it appends them to the color block and outputs them without making any color change.

The complete construction of the advice sequence is given in Section 4. In Section 5 we show that the construction of the advice sequence guarantees that ALG can always choose a color block to be output (i.e., there is always a block of type COMPLETE, READY or LIST in the buffer of ALG), and that ALG can always output all elements of a complete color block without any color change in between (Theorem 1). That means in particular that the cost of the algorithm is upper bounded by the number of color blocks in the advice sequence.

4 Constructing the Advice Sequence

Overview. The advice sequence $A_\varepsilon(\sigma)$ is constructed *offline*, based on an *optimal* solution OPT for the instance σ of the reordering buffer management problem. The idea of the construction is as follows. We initially assign each input element e type t_e based on the way OPT handles e. Let c be the color of e. If OPT makes a color change right after outputting e, e is assigned type COMPLETE. Otherwise, if the whole color block containing e is kept in the buffer of OPT until the next element of color c is read from the input sequence, e gets type WAIT. Otherwise, e gets type LIST. This ensures the following invariant.

Invariant 1 *For each color block, its elements are output by* OPT *in a single block (i.e., without making a color change).*

For each color block B which contains elements of type LIST, we want to assign a value v_B, which is the initial position of the block on the waiting list. The blocks in the waiting list are ordered according to the order in which they are output by OPT. For each color block, if the advice included the exact position of the block in the waiting list based on the output sequence of opt, the algorithm ALG would output an optimal solution. However, for some short color blocks, we cannot encode their position on the waiting list. To deal with this problem, we will modify the advice data so as to decrease the number of blocks which

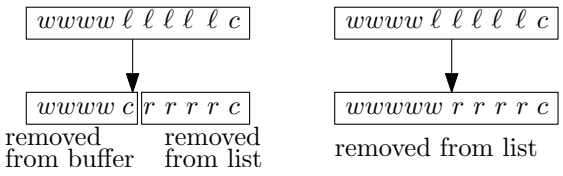

Fig. 1. The procedures Split(B) (on the left) and Postpone(B) (on the right). None of the resulting blocks remain on the waiting list. The letters w, l, r, c are used to denote elements of types WAIT, LIST, READY and COMPLETE, respectively. The frames represent color blocks.

are in the waiting list at any time. This modification increases the number of color blocks (i.e., the cost of ALG becomes larger than the cost of OPT), and introduces elements of type READY. Blocks containing elements of type READY are not inserted into the waiting list. We now give an overview of the procedures used to modify the advice data.

Procedure Remove(B) Remove the block B from the waiting list. For each block B' that was inserted into the waiting list at a time later than B, and at a position (in the list) behind B, decrease $v_{B'}$ by one.

Procedure Split(B) Run Remove(B). Reassign type COMPLETE to the first element of block B which had type LIST, and reassign type READY to the remaining elements of B which had type LIST assigned (see Figure 1, left). Note that possibly some of these elements have not been read from the input yet.

The block B has been split into two blocks, both ending with an element of type COMPLETE. The elements of the first block (called the *early* block) are removed from the buffer. The second block is called a *late* block. This block is kept in the buffer (but not on the waiting list).

Procedure Postpone(B) Run Remove(B). Reassign type WAIT to the first element of B which originally had type LIST assigned, and reassign type READY to the remaining elements of B which had type LIST assigned (see Figure 1, right). Possibly some of these elements have not been read from the input yet. Blocks processed in this way are called *postponed blocks*.

The procedure Split(B) makes ALG evict the elements from the first block of B (i.e., the early block) earlier than they were evicted by OPT, generating free space in the buffer of ALG. This allows some blocks to stay in the buffer until the next element of the block is read from the input, and only then to be output by ALG. These are the blocks for which we run procedure Postpone(B). We will specify later which blocks will be treated in this way. A block B that is removed from the waiting list by these procedures will never be inserted into the waiting list of ALG. This is the reason we update the values $v_{B'}$ of other blocks

on the waiting list when applying procedure Remove(B). We have the following observation.

Observation 1 *The procedures Split(B) and Postpone(B) maintain Invariant 1.*

We are now ready to describe the details of the construction of the advice sequence $A_\varepsilon(\sigma)$. The input elements from the sequence σ are processed one by one, using a buffer of size k (which we call an *advice buffer*) and a waiting list. Notice that at this point we are only creating an advice sequence, that is, assigning type t_e and value v_e for each input element e, and not creating an output sequence. In particular, "removing elements from the buffer" does not mean "appending elements to the output sequence".

Processing an input element Whenever there is empty space in the buffer, we read the next element e from the input sequence σ. We proceed as follows. First, we assign the initial type of the element e as described above, based on the way OPT handles e. Then, if e is the first element in its block which is of type LIST, i.e., its color block changes type to LIST, we insert the color block of e at the appropriate position, based on the optimal output sequence OPT, into the waiting list. After this operation, if the waiting list has too many blocks of length similar to the one added, we remove all such blocks from the waiting list by applying procedure Split(B) to some of the blocks, and procedure Postpone(B) to the remaining blocks of the given length. We use the following definition.

Definition 1. *The* class *of a block B on the waiting list is $\lfloor \log |B| \rfloor$, where $|B|$ is the number of items of this block up to and including the first element of type* LIST.

We now give the detailed description of actions performed for a newly read element e of color c. Here C is another constant which will depend on ε.

1. (*Type assignment*) We assign the element e type t_e based on the way OPT handles e (see Figure 2).
   (a) If OPT makes a color change right after outputting e, e is assigned type COMPLETE.
   (b) Else, if e has already been assigned type READY due to some previous Split or Postpone operation (i.e., before e has been read into the buffer), it keeps the type READY.
   (c) Else, if the whole color block containing e is kept in the buffer of OPT until the next element element of color c is read from the input sequence, e gets type WAIT.
   (d) Else, e gets type LIST.

   Note that the type of the element e can be modified later by applying procedure Split(B) or Postpone(B) to the color block B of e.
2. (*Insertion into the waiting list*) If e is the *first* element of its color block of type LIST, insert the color block B of e into the waiting list at position v_B

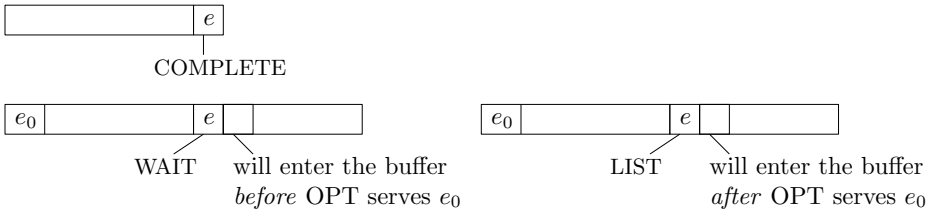

Fig. 2. Possible initial types of elements (excluding the special case of READY). The type LIST indicates that OPT starts serving this color block before all elements have been read into the buffer; the remaining elements enter the buffer while the color block is being served (and while other elements of the color block are being removed).

(derived from the solution OPT). Let i be the class of block B. Count the number n_i of color blocks of class i which are currently in the waiting list.

If $n_i = C$, let B_1 and B_2 be the two last color blocks of class i in the waiting list. Run Split(B_1) and Split(B_2). For each other block B_0 of class i on the waiting list, run Postpone(B_0). By performing these operations, we ensure that ALG will never insert these C blocks into the waiting list.

Note that in step 2 some blocks may be split, with the resulting early blocks being removed from the buffer.

Whenever the advice buffer becomes full, and there are no elements of the current color block in the buffer, we choose a new color block B to be processed (i.e., removed from the buffer), according to the rules of ALG. The following lemma shows that the rules of ALG can always be successfully applied.

Lemma 1. *While creating the advice sequence, if blocks are removed from the buffer, according to the rules of* ALG *and Step 2 above, then starting from time k there is always an element of type different than* WAIT *in the buffer.*

Let B be the color block chosen from the full advice buffer according to the rules of ALG. If B is not on the waiting list, we simply remove it from the buffer. If B is on the waiting list, we have to consider two cases.

1. The value v_B is smaller than $D^{|B|}$, i.e., it can be encoded using $|B|$ values v_e of the elements of the block. We set appropriately the values v_e, and we remove B from the buffer and from the waiting list. (Note that this time we do not apply the procedure Remove(B), i.e., we do not modify any values $v_{B'}$ of blocks B'.)
   Only blocks processed in this way will ever be inserted by ALG into the waiting list.
2. The value v_B is at least $D^{|B|}$. Run Split(B). In this case, only the early block obtained from B is removed from the buffer.

Notice that, as above, in determining the length of B in Step 1, we count only the elements up to and including the first element of type LIST. The reason is that these are the only elements of B that have been inserted into the buffer before B has to be inserted into the waiting list, i.e., before the value of B has to be computed.

Note also that the initial value of v_B might decrease (due to calls to the procedure Remove when B is already in the waiting list) before B is processed by ALG. Specifically, it does not matter if v_B was initially too high, as long as it drops below $D^{|B|}$ before the block is chosen to be processed.

5 Analysis of ALG

Lemma 1 leaves open the possibility that ALG may be able to serve only *part* of a block. This could happen if not all elements of the (complete) block enter the buffer while ALG is serving its color. In such a case, ALG would have to return to this color several times, and its cost would be higher than the number of color blocks given by the advice sequence $A_\varepsilon(\sigma)$. At the beginning of this section, we will show that this does not happen, and ALG always outputs complete color blocks. Then, we will bound the competitive ratio of ALG.

First, we present two technical lemmas.

Lemma 2. *In the advice sequence $A_\varepsilon(\sigma)$, types of the consecutive elements from one complete color block always form an expression*

$$(\text{WAIT})^* \left((\text{LIST})^* \cup (\text{READY})^*\right) \text{COMPLETE}.$$

Lemma 3. *Let $c_1, \ldots, c_\ell$ be a collection of consecutive elements of color c, output by* OPT *with no color change in between, and let t be the time when* OPT *outputs the first element c_1 of the collection. If* ALG *outputs c_1 at some time $t' \geq t$, and it has not output any c_i before time t', it can output all elements $c_1, \ldots, c_\ell$ with no color change in between.*

We can now consider all types of blocks which ALG starts outputting before they become complete, and show that OPT cannot keep the elements of such blocks longer in the buffer. Applying Lemma 3 gives us the following results.

Lemma 4. *When the algorithm* ALG *starts outputting a color block, it can finish it with no color changes.*

Theorem 1. *The algorithm* ALG *always finds a color block to be output, and outputs only complete color blocks.*

In the remaining part of this section, we will bound the competitive ratio of ALG, if ALG uses the advice sequence $A_\varepsilon(\sigma)$. For this, we will need an upper bound on the cost of ALG, as well as a lower bound on the cost of OPT. From the construction of the advice sequence, we know the following.

Observation 2 *The cost of* ALG *is upper bounded by the number of color blocks given by the advice sequence* $A_\varepsilon(\sigma)$.

Observation 3 *The cost of* OPT *is lower bounded by the number of color blocks other than the late blocks given by the advice sequence* $A_\varepsilon(\sigma)$.

To bound the competitive ratio, it is enough to bound the number of late color blocks in $A_\varepsilon(\sigma)$, compared to the number of color blocks of other types. Late blocks can be generated in two ways: while inserting new elements into the waiting list, and while processing a color block at the head of the waiting list. We call the sets of late blocks generated during these two operations by LATE_I and LATE_P, respectively.

Lemma 5. *We have* $|\text{LATE}_I| \leq \frac{2}{C}(|\text{POSTPONED}| + |\text{EARLY}|)$, *where* POSTPONED *and* EARLY *denote the sets of postponed and early blocks for* $A_\varepsilon(\sigma)$.

Lemma 6. *We have* $|\text{LATE}_P| \leq \frac{C}{D-1}|\text{LISTED}|$, *where* LISTED *denotes the set of color blocks which contain elements of type* LIST.

Combining the two Lemmas above and setting $C = \lceil 2/\varepsilon \rceil$ and $D = \lceil C/\varepsilon \rceil + 1$ gives us the main theorem of the paper.

Theorem 2. *For any* $\varepsilon > 0$, *there is a* $(1+\varepsilon)$*-competitive algorithm for the reordering buffer management problem which uses only* $O(\log(1/\varepsilon))$ *advice bits per input item.*

6 Optimality Lower Bound

In this section, we show that a deterministic algorithm requires $\Omega(\log k)$ bits of advice per request in order to be optimal. Throughout this section, we assume without loss of generality that an algorithm will only perform a color switch when there are no more items of the current color in the buffer, and the buffer is either full or contains the last request. Let k be the size of the buffer, and let

$$\sigma = \big\langle c_1, c_2 \ldots, c_k, \pi_1(1), c_{k+1}, \pi_1(2), c_{k+2} \ldots, \pi_1(k), c_{2k}, \pi_2(1), \pi_2(2), \ldots, \pi_2(k) \big\rangle,$$

where π_1 is a permutation of the colors $c_1, \ldots, c_k$, and π_2 is a permutation of the colors $c_{k+1}, \ldots, c_{2k}$. Note that $|\sigma| = 4k$.

As long as the buffer of an optimal algorithm remains full, the algorithm must switch to the color c_i when the next element waiting to enter the buffer has color c_i. In particular, the first k consecutive colors output by the algorithm must be $\pi_1(1), \ldots, \pi_1(k)$. More formally,

Lemma 7. *Given* σ, *let* S *be the sequence of colors* $\pi_1(1), \ldots, \pi_1(k)$. *Let* $\mathcal{C}$ *be the set of all the possible optimal sequences of color switches for* σ. *The* k*-prefix of all the sequences in* $\mathcal{C}$ *is* S.

Using Lemma 7, we can prove the main theorem of this section.

Theorem 3. *At least* $\frac{\log k}{8}$ *bits of advice per request are required for a deterministic algorithm with advice to be optimal.*

References

1. Adamaszek, A., Czumaj, A., Englert, M., Räcke, H.: Almost tight bounds for reordering buffer management. In: Fortnow, L., Vadhan, S.P. (eds.) STOC. pp. 607–616. ACM (2011)
2. Asahiro, Y., Kawahara, K., Miyano, E.: Np-hardness of the sorting buffer problem on the uniform metric. Discrete Applied Mathematics 160(10-11), 1453–1464 (2012)
3. Avigdor-Elgrabli, N., Rabani, Y.: An improved competitive algorithm for reordering buffer management. In: Charikar, M. (ed.) SODA. pp. 13–21. SIAM (2010)
4. Avigdor-Elgrabli, N., Rabani, Y.: A constant factor approximation algorithm for reordering buffer management. In: Khanna, S. (ed.) SODA. pp. 973–984. SIAM (2013)
5. Avigdor-Elgrabli, N., Rabani, Y.: An optimal randomized online algorithm for reordering buffer management. CoRR abs/1303.3386 (2013)
6. Blandford, D.K., Blelloch, G.E.: Index compression through document reordering. In: DCC. pp. 342–351. IEEE Computer Society (2002)
7. Böckenhauer, H.J., Komm, D., Královic, R., Královic, R.: On the advice complexity of the k-server problem. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP (1). Lecture Notes in Computer Science, vol. 6755, pp. 207–218. Springer (2011)
8. Böckenhauer, H.J., Komm, D., Královic, R., Královic, R., Mömke, T.: On the advice complexity of online problems. In: Dong, Y., Du, D.Z., Ibarra, O.H. (eds.) ISAAC. Lecture Notes in Computer Science, vol. 5878, pp. 331–340. Springer (2009)
9. Chan, H.L., Megow, N., Sitters, R., van Stee, R.: A note on sorting buffers offline. Theor. Comput. Sci. 423, 11–18 (2012)
10. Dobrev, S., Kralovic, R., Pardubská, D.: How much information about the future is needed? In: Geffert, V., Karhumäki, J., Bertoni, A., Preneel, B., Návrat, P., Bieliková, M. (eds.) SOFSEM. Lecture Notes in Computer Science, vol. 4910, pp. 247–258. Springer (2008)
11. Emek, Y., Fraigniaud, P., Korman, A., Rosén, A.: Online computation with advice. Theor. Comput. Sci. 412(24), 2642–2656 (2011)
12. Englert, M., Räcke, H., Westermann, M.: Reordering buffers for general metric spaces. Theory of Computing 6(1), 27–46 (2010)
13. Englert, M., Westermann, M.: Reordering buffer management for non-uniform cost models. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP. Lecture Notes in Computer Science, vol. 3580, pp. 627–638. Springer (2005)
14. Gutenschwager, K., Spiekermann, S., Voß, S.: A sequential ordering problem in automotive paint shops. Internat. J. Production Research 42(9), 1865–1878 (2004)
15. Hromkovic, J., Královic, R., Královic, R.: Information complexity of online problems. In: Hlinený, P., Kucera, A. (eds.) MFCS. Lecture Notes in Computer Science, vol. 6281, pp. 24–36. Springer (2010)
16. Komm, D., Královic, R.: Advice complexity and barely random algorithms. RAIRO - Theor. Inf. and Applic. 45(2), 249–267 (2011)
17. Krokowski, J., Räcke, H., Sohler, C., Westermann, M.: Reducing state changes with a pipeline buffer. In: Girod, B., Magnor, M.A., Seidel, H.P. (eds.) VMV. p. 217. Aka GmbH (2004)
18. Räcke, H., Sohler, C., Westermann, M.: Online scheduling for sorting buffers. In: Möhring, R.H., Raman, R. (eds.) ESA. Lecture Notes in Computer Science, vol. 2461, pp. 820–832. Springer (2002)
19. Renault, M.P., Rosén, A.: On online algorithms with advice for the k-server problem. Theory of Computing Systems pp. 1–19 (2012)