Algorithmica manuscript No.
(will be inserted by the editor)

List Factoring and Relative Worst Order Analysis

Martin R. Ehmsen · Jens S. Kohrt · Kim S. Larsen

Received: date / Accepted: date

Abstract Relative worst order analysis is a supplement or alternative to competitive analysis which has been shown to give results more in accordance with observed behavior of online algorithms for a range of different online problems. The contribution of this paper is twofold. As the first contribution, it adds the static list accessing problem to the collection of online problems where relative worst order analysis gives better results. List accessing is a classic data structuring problem of maintaining optimal ordering in a linked list. It is also one of the classic problems in online algorithms, in that it is used as a model problem, along with paging and a few other problems, when trying out new techniques and quality measures. As the second contribution, this paper adds the non-trivial supplementary proof technique of list factoring to the theoretical toolbox for relative worst order analysis. List factoring is perhaps the most successful technique for analyzing list accessing algorithms, reducing the complexity of the analysis of algorithms on full-length lists to lists of length two.

A preliminary version of this paper appeared in the proceedings of the 8th Workshop on Approximation and Online Algorithms. This work was supported in part by the Danish Natural Science Research Council.

Martin R. Ehmsen
Department of Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark

Jens S. Kohrt
Department of Mathematics and Computer Science and CP^3-Origins, University of Southern Denmark, Odense, Denmark

Kim S. Larsen
Department of Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark
Tel.: +45 6550 2328
E-mail: kslarsen@imada.sdu.dk

Keywords online algorithms · list accessing · relative worst order analysis · list factoring

1 Introduction

The static list accessing problem [28, 4] is a well-known problem in online algorithms. Many deterministic as well as randomized algorithms are known, and these have been investigated theoretically as well as experimentally. See [10] for a discussion of the importance of the problem in relation to dictionary implementation, connections to paging, and applications in compression algorithms. For readers unfamiliar with the standard algorithms for list accessing or relative worst order analysis, we refer to the rigorous definitions in Section 2 and Section 3.

The most standard performance measure for online algorithms is competitive analysis [20, 28, 22]. The starting point for our work was the discrepancy between the findings obtained regarding list accessing when analyzed using competitive analysis and when investigated through experimental work. Competitive analysis finds that Move-To-Front is optimal, with a competitive ratio of two, while Frequency-Count and Transpose have lower bounds on their competitive ratio which grow linearly with the length of the list [10]. The point is that the difference is dramatic and that Frequency-Count is placed in the "bad" group.

In contrast, experimental work [9] on Move-To-Front and Frequency-Count suggests that these algorithms are almost equally good and both are far better than Transpose. That paper [9] reports on numerous algorithms and experiments directly investigating list accessing or applications of list accessing. The authors of [9] conducted experiments on real data. They pointed out that real data is always biased in some direction, and consequently for concrete applications, real data properties may give specific insight. They concluded that Frequency-Count is among the top algorithms in both of the two data scenarios they consider [9, p. 58]. MTF is not in the top group, but variants of MTF are. Also, Transpose is never in the top group. Thus, considering this experimental data, it seems that Frequency-Count is misplaced by competitive analysis.

Other experiments [7] reach similar conclusions. The authors of [7] find that Transpose is consistently significantly worse than both MTF and Frequency-Count. With regards to MTF and Frequency-Count, the former is most often a little better than Frequency-Count, but the difference is smaller and in 2 out of 9 tests, Frequency-Count is better. This also indicates that if these algorithms should be categorized in two groups, Transpose should be in the "bad" group and MTF and Frequency-Count together in the better group.

To a large extent driven by the paging problem [10] and the difficulties encountered in theoretically separating various algorithm proposals, many alternative performance measures have been developed to supplement standard competitive analysis. Examples include [30, 8, 23, 24, 11, 5]; see [16] for a survey.

Some of these measures are tailored towards a specific online problem, whereas others are more generally applicable; see [13] for a comparative study of these measures on a simple problem.

Of these alternatives to competitive analysis, relative worst order analysis [11, 12] is the measure that has been applied to the largest variety of online problems. Results that are in accordance with experiments have been derived for a range of fairly different online problems in situations where competitive analysis has given the “wrong” answer. Online problems of this nature include (but are not limited to) the following:

- For classical bin packing, Worst-Fit is better than Next-Fit [11].
- For dual bin packing, First-Fit is better than Worst-Fit [11].
- For paging, LRU is better than FWF (Flush-When-Full) and look-ahead helps [12].
- For scheduling, minimizing makespan on two related machines, a post-greedy algorithm is better than scheduling all jobs on the fast machine [18].
- For bin coloring [25], a natural greedy-type algorithm is better than just using one open bin at a time [17].
- For seat reservation, First-Fit is better than Worst-Fit [14] with regards to proportional price.

We apply relative worst order analysis to the static list accessing problem. We first extend the list factoring technique [9, 3] known from competitive analysis to relative worst order analysis. We then apply the technique to the three deterministic online list accessing algorithms Move-To-Front, Time-Stamp, and Frequency-Count. We show that these algorithms are equally good and much better than Transpose when analyzed using relative worst order analysis, thereby obtaining results that are in accordance with the cited experimental work.

Adding static list accessing to the collection of problems above where relative worst order analysis gives better or more nuanced results than competitive analysis is a step in documenting to what extent relative worst order analysis is generally applicable. However, we find it more interesting that relative worst order analysis can be equipped with a powerful supplementary proof technique such as list factoring. To our knowledge, relative worst order analysis is the first of the alternative performance measures to be equipped with a list factoring lemma.

Some of the deterministic list accessing algorithms are quite old. It is difficult to pin-point the origin of Frequency-Count, since it is intimately related to probability theoretical considerations, and it is not clear when it started being viewed as an algorithm. Move-To-Front and Transpose were formulated in [26]. Time-Stamp [1] is a deterministic algorithm that arose as a special case of a family of randomized algorithms.

We also consider BIT [27] and Randomized-Move-To-Front[1], both of which are randomized algorithms. Deterministic and randomized online algorithms are often compared informally, but it is not clear how much sense it makes to compare a worst-case guarantee with an average-case performance. We compare the two randomized algorithms to each other and find them incomparable whereas competitive analysis slightly favors the former (often referred to as a "surprising" result [10, p. 27]), showing that BIT is $\frac{7}{4}$-competitive and Randomized-Move-To-Front is 2-competitive, in both cases against an oblivious adversary [27,19].

For early related work, we refer the reader to [4,10]. Newer work obtains separations between list accessing algorithms by analyzing these with respect to some measure of locality of reference [2,6,15].

2 List Accessing

In the *static list accessing problem* [28,4], we have a fixed collection of items arranged in a linear list, $\mathcal{L} = (a_1, a_2, \ldots, a_\ell)$, of length ℓ. The request sequence, I, consists of requests of *access* to items in the list, and the accesses must be served in an online manner. To access an item, one must start searching from the front of the list, inspecting each item until the correct item is found. Thus, if we are searching for some item x, we first compare x with a_1. If they are the same, we stop. Otherwise, we proceed to compare x with a_2. The process continues until x is found.

Thus, accessing an item currently at index j also involves comparisons to $j-1$ other items in front of it. In treating one request, the final comparison, which determines that the requested item has been found, is referred to as a *positive* comparison, whereas the comparisons to the $j-1$ items in front of it are referred to as *negative* comparisons.

The *cost* of accessing an item depends on its position (index) in the list. In the *full cost model*, one pays for every comparison, so accessing the item at index j costs j. In the *partial cost model*, the final positive comparison is not counted, so accessing an item currently at index j costs $j-1$. Since every access to an item in the list must end with a positive comparison, it is only the total number of negative comparisons that can be affected by the choice of algorithm. The partial cost model measures exactly this parameter. Also, when using this model, proofs often become more elegant, making this a common choice during the analysis, e.g., when using the list factoring technique. Since we know the number of requests, one can easily compute the total costs in one model from the result found in the other.

After accessing an item, it can be moved to any position further towards the front of the list without any additional cost. Such a move can be seen as a number of transpositions of the accessed item with items preceding it

[1] In the many papers that discuss Randomized-Move-To-Front, we have not been able to find a reference to the paper with the first definition of the algorithm. However, [7] cites personal communication with J. Westbrook from 1996 regarding properties of the algorithm.

in the list. The transpositions used to perform such a move are called *free*. Furthermore, at any time, an algorithm may exchange two adjacent items in the list at a cost of one. Such a transposition is denoted a *paid* transposition. The objective of a list accessing algorithm is to use free and paid transpositions in order to minimize the overall cost of serving the request sequence. Further discussion of the modelling issues can be found in [10].

In this paper, we only consider algorithms that do not use paid transposition, and unless otherwise noted, we use the full cost model. Many of the proofs are carried out using the partial cost model.

Many different algorithms have been proposed for the list accessing problem. Some of the most well-known deterministic paging algorithms are the following.

MTF (Move-To-Front): After accessing the requested item, MTF moves the item to the front of the list.

FC (Frequency-Count): After accessing the requested item, FC moves the item forward in the list such that the resulting list is in sorted order with respect to the frequency with which the items have been accessed, i.e., for every item, FC maintains a counter which is incremented on an access to the item and the list is sorted in non-increasing order of the counters. FC only moves the accessed item forward the least number of positions necessary to maintain the sorted order.

TS (Time-Stamp): After accessing item a_i, it is inserted in front of the first item a_j (from the front of the list) that precedes a_i in the list and was accessed at most once since the last access to a_i. The algorithm does nothing if there is no such item a_j or if a_i is accessed for the first time.

TRANS (Transpose): After accessing the requested item, it is transposed with the item in front of it in the list. If the item is already at the front of the list, it stays there.

In addition to the above deterministic algorithms, we also consider the following well-known randomized algorithms.

BIT: For each item in the list, BIT [27] maintains a bit. Before processing a request sequence, BIT initializes the bits independently and uniformly at random. On a request for an item, BIT first complements the item's bit. If the bit is then one, the item is moved to the front of the list. Otherwise, BIT does not move the item.

RMTF (Randomized-Move-To-Front): After each access to a requested item, RMTF moves the item to the front of the list with probability $\frac{1}{2}$.

As an example, in the full cost model consider MTF on the input sequence $I = \langle b, a, a, c, b \rangle$ and the initial list $\mathcal{L} = (a, b, c)$. The first request to b has a cost of two, since b is initially at index two. The algorithm MTF then changes the list to (b, a, c), i.e., it moves the requested item to the front of the list. By definition of the list accessing problem, it may do so at no cost since b is the requested item. In treating these five requests, the list starts in state (a, b, c) and then goes through the states (b, a, c), (a, b, c), (a, b, c), (c, a, b), and (b, c, a) at a cost of 2, 2, 1, 3, and 3, respectively, summing up to a total cost of 11.

3 Relative Worst Order Analysis

Relative worst order analysis was first introduced in [11] in an effort to combine the desirable properties of the max/max ratio [8] and the random-order ratio [23]. The measure was later refined in [12].

Instead of comparing online algorithms to an optimal offline algorithm (and then comparing their competitive ratios), two online algorithms are compared directly. However, instead of comparing their performance on the exact same request sequence, they are compared on their respective worst permutations of the same sequence.

Formally, if I is a request sequence of length n and σ is a permutation on n elements, then $\sigma(\mathrm{I})$ denotes I permuted by σ. Let A be a list accessing algorithm and let A(I) denote the cost of running A on I. In the examples below, we use the full cost model, but point out that the definitions given in this section are independent of cost model, and therefore also can be used with the partial cost model. Define $\mathrm{A_W(I)}$ to be the performance of A on a worst possible permutation of I with respect to A, i.e., $\mathrm{A_W(I)} = \max_\sigma\{\mathrm{A}(\sigma(\mathrm{I}))\}$.

Continuing the example from the previous section, where we considered MTF on $I = \langle b, a, a, c, b\rangle$ with the initial list $\mathcal{L} = (a, b, c)$, the total cost was 11, so $\mathrm{MTF}(I) = 11$. However, a worst permutation of I would be $I_{\mathrm{MTF}} = \langle c, b, a, b, a\rangle$, which has a cost of 13. Thus, $\mathrm{MTF}_W(I) = 13$.

For any pair of algorithms A and B, we define

$$\mathrm{c_u(A,B)} = \inf\{c \mid \exists b\colon \forall\, \mathrm{I}\colon\ \mathrm{A_W(I)} \le c\,\mathrm{B_W(I)} + b\} \quad \text{and}$$
$$\mathrm{c_l(A,B)} = \sup\{c \mid \exists b\colon \forall\, \mathrm{I}\colon\ \mathrm{A_W(I)} \ge c\,\mathrm{B_W(I)} - b\}.$$

Note that the additive constant b must be independent of the request sequence I, but is allowed to depend on the initial state of the list that A is working on. This is similar to how the additive constant is treated when using competitive analysis.

Intuitively, $\mathrm{c_l}$ and $\mathrm{c_u}$ can be thought of as tight lower and upper bounds, respectively, on the performance of A relative to B.

If $\mathrm{c_l(A,B)} \ge 1$ or $\mathrm{c_u(A,B)} \le 1$, the algorithms are said to be *comparable*. For instance, if $\mathrm{c_u(A,B)} \le 1$, this means that there exists a constant $c \le 1$ such that, possibly ignoring an additive constant, the cost of A is at most c times the cost of B for all input sequences. Thus, again ignoring the additive constant, A never has a higher cost than B. This is the most important information we want to compute concerning the relationship between two algorithms.

If two algorithms are comparable, then the *relative worst order ratio*, $\mathrm{WR_{A,B}}$, of algorithm A to algorithm B is defined. Otherwise, $\mathrm{WR_{A,B}}$ is undefined.

$$\text{If } \mathrm{c_u(A,B)} \le 1, \text{ then } \mathrm{WR_{A,B}} = \mathrm{c_l(A,B)}, \text{ and}$$
$$\text{if } \mathrm{c_l(A,B)} \ge 1, \text{ then } \mathrm{WR_{A,B}} = \mathrm{c_u(A,B)}.$$

Having already established that the algorithms are comparable, and, thus, one algorithm at least as good as the other, the relative worst order ratio is a bound on how much better the one algorithm can be.

If the ratio is strictly smaller or larger than one, then one algorithm is strictly better than the other: if $\mathrm{WR}_{\mathrm{A,B}} < 1$, then A is better than B, and if $\mathrm{WR}_{\mathrm{A,B}} > 1$, then B is better than A. We introduce a combined term for this: if A and B are comparable and A is better than B according to the relative worst order ratio, then A and B are *comparable in* A*'s favor*. Finally, if the ratio is one, then the algorithms perform identically according to relative worst order analysis.

In [11, 12], it was shown that the relative worst order ratio is a transitive measure, i.e., the relative worst order ratio defines a partial ordering of the algorithms for a given problem.

Though we discuss results obtained using competitive analysis, all our results are obtained using relative worst ordering analysis.

4 List Factoring

The list factoring technique was first introduced by Bentley and McGeoch [9] and later extended and improved in a series of papers [21, 29, 1, 3]. It reduces the analysis of list accessing algorithms to lists of size two. Previously the technique was developed and applied only in the context of competitive analysis, where it can be used to prove upper bounds on the competitive ratio [10]. In this section, we show that list factoring can also be applied in the context of relative worst order analysis to separate online algorithms and prove upper bounds.

For the definitions in this section, let A denote any online list accessing algorithm that does not use paid transpositions. We are going to consider the partial cost model where accessing the ith item in the list costs $i-1$. For any request sequence I, we use the standard notation (from [10], for instance) of $\mathrm{A}^\star(\mathrm{I})$ to denote the cost A incurs while processing I in the partial cost model.

Consider the list when A is about to process the ith request I_i and define

$$\mathrm{A}^\star(a_j, i) = \begin{cases} 1 & \text{if } a_j \text{ is in front of } I_i \text{ in the list} \\ 0 & \text{otherwise (including } a_j = I_i\text{)} \end{cases}$$

for all items a_j in the list.

We also define

$$\mathrm{A}^\star_{ab}(\mathrm{I}) = \sum_{i: I_i \in \{a,b\}} \left(\mathrm{A}^\star(a,i) + \mathrm{A}^\star(b,i)\right).$$

$\mathrm{A}^\star(a_j, i)$ can be viewed as representing the cost due to a_j of accessing the ith item, I_i, since if a_j is in front of I_i just before the request to I_i is processed, a comparison with a_j will result. Similarly, $\mathrm{A}^\star_{ab}(\mathrm{I})$ then represents the part of the total cost of processing the entire request sequence which is due to a and b being in front the requested items, but only considering requests to a and b in the sequence. Thus, for each term in the sum, either $\mathrm{A}^\star(a,i)$ or $\mathrm{A}^\star(b,i)$ is zero. Thus, the sum, $\mathrm{A}^\star(a,i) + \mathrm{A}^\star(b,i)$ is one if I_i is a request to a and b is in front of a at the time or if I_i is a request to b and a is in front of b at the time. Otherwise, the term is zero.

The above leads to the observation, also made in [10], that we can express the total cost of A on a sequence I in the partial cost model by considering $\mathrm{A}^\star_{ab}(\mathrm{I})$ over all pairs of a and b, i.e.,

$$\mathrm{A}^\star(\mathrm{I}) = \sum_{\{a,b\}\subseteq\mathcal{L}, a\neq b} \mathrm{A}^\star_{ab}(\mathrm{I}).$$

Let I_{ab} be the projection of I over a and b, i.e., the sequence obtained from I by deleting all requests to items other than a or b. Similarly, let $\mathcal{L}_{ab}$ be the initial list with items other than a and b deleted.

An algorithm A is said to have the *pairwise property*, if for all pairs, a and b, of two items in $\mathcal{L}$, we have

$$\mathrm{A}^\star_{ab}(\mathrm{I}) = \mathrm{A}^\star(\mathrm{I}_{ab}).$$

where, on the left-hand side, A is processing I on the full list $\mathcal{L}$, whereas on the right-hand side, A is processing I_{ab} on the list $\mathcal{L}_{ab}$.

Thus, on the left-hand side, A processes the full request sequence, but we are only summing the costs having to do with the relative position of a and b and only on requests to one of these items, i.e., the cost of processing a request is one if we are accessing one of these items and the other is in front. Otherwise the cost of that access is zero. On the right-hand side, we process the projected input sequence on the projected list and count all costs (still in the partial cost model).

Intuitively, the equality then holds when the algorithm's treatment of an accessed item is independent of other items. The algorithm MTF behaves that way, whereas Transpose does not. To see this, assume that at some point, the full list is in the state (a, c, b). Then in accessing b, Transpose would move b forward to obtain (a, b, c). However, in the projected list of (a, b), Transpose would change the state to (b, a). Thus, the relative positions of a and b end up being different in the full list and the projected list. This means that the costs of the two sides of the equation defining the pairwise property end up being different, and then that equality does not hold.

In competitive analysis, this setup can be used to prove upper bounds on the competitive ratio (in the partial cost model) of algorithms that have the pairwise property. In addition, if the algorithms are also cost independent, then the ratio carries over to the full cost model [10]. An algorithm is *cost independent* if the decisions it makes are independent of the cost. There are online algorithms (for other online problems) that use accumulated cost as part of the decision making in order to obtain a particular end result. Such algorithms would simply function differently if the cost model was changed. Of course one might be able to modify the algorithm with regards to the modified cost, but it may be non-trivial if it is not just the total cost that is used, but for instance a ratio of accumulated cost compared to accumulated cost by an optimal offline algorithm.

For relative worst order analysis, we show that the setup above can also be used to separate algorithms and prove upper bounds. Consider an algorithm A that has the pairwise property. It follows that

$$\mathrm{A}^\star_\mathrm{W}(\mathrm{I}_{ab}) = \max_\sigma \mathrm{A}^\star(\sigma(\mathrm{I}_{ab})) = \max_\sigma \mathrm{A}^\star((\sigma(\mathrm{I}))_{ab}) = \max_\sigma \mathrm{A}^\star_{ab}(\sigma(\mathrm{I})).$$

The three equalities follow from the definition of a worst order, simple properties of permutations, and the pairwise property, respectively.

We now say that A has the *worst order projection property*, if and only if for all sequences I, there exists a worst ordering $\sigma_\mathrm{A}(\mathrm{I})$ of I with respect to A, such that for all pairs $\{a,b\} \subseteq \mathcal{L}$ $(a \neq b)$, $\sigma_\mathrm{A}(\mathrm{I})_{ab}$ is a worst ordering of I_{ab} with respect to A on the initial list $\mathcal{L}_{ab}$.

It is most illustrative to show how it could be possible that an algorithm does not have the property. Consider Transpose on the input sequence $\langle a,b,c,c\rangle$ with initial list (a,b,c). A worst ordering is $\langle c,c,b,a\rangle$ which has a cost of 11. Considering the projection $\langle c,c,a\rangle$ on a and c on the projected list (a,c), we get a cost of 2, but then $\langle c,a,c\rangle$ is a worse ordering, since it has a cost of 3. Thus, the projection is not the worst ordering.

Using the above, we obtain a lemma similar to the Factoring Lemma for competitive analysis [10].

Lemma 1 *Let* A *and* B *be two online list accessing algorithms that do not use paid transpositions and that have the pairwise property and the worst order projection property, and let* $\mathcal{L}$ *be a list. If there exists constants* c *and* b_1 *such that for every pair* $\{a,b\} \subseteq \mathcal{L}$ $(a \neq b)$*, and for every request sequence* I, $\mathrm{A}^\star_\mathrm{W}(\mathrm{I}_{ab}) \leq c\,\mathrm{B}^\star_\mathrm{W}(\mathrm{I}_{ab}) + b_1$*, then there exists a constant* b_2 *such that for every request sequence* I, $\mathrm{A}^\star_\mathrm{W}(\mathrm{I}) \leq c\,\mathrm{B}^\star_\mathrm{W}(\mathrm{I}) + b_2$.

In addition, if A *and* B *are cost independent and* $c \geq 1$*, then* $\mathrm{A}_\mathrm{W}(\mathrm{I}) \leq c\,\mathrm{B}_\mathrm{W}(\mathrm{I}) + b_2$.

Proof Consider any algorithm A satisfying the hypothesis. Then

$$\begin{aligned}\mathrm{A}^\star_\mathrm{W}(\mathrm{I}) &= \max_\sigma \mathrm{A}^\star(\sigma(\mathrm{I})) = \max_\sigma \sum_{\{a,b\}\subseteq\mathcal{L},a\neq b} \mathrm{A}^\star_{ab}(\sigma(\mathrm{I})) \\ &= \sum_{\{a,b\}\subseteq\mathcal{L},a\neq b} \max_\sigma \mathrm{A}^\star_{ab}(\sigma(\mathrm{I})) = \sum_{\{a,b\}\subseteq\mathcal{L},a\neq b} \mathrm{A}^\star_\mathrm{W}(\mathrm{I}_{ab}).\end{aligned}$$

Now consider two algorithms A and B satisfying the hypothesis. We get

$$\begin{aligned}\mathrm{A}^\star_\mathrm{W}(\mathrm{I}) &= \sum_{\{a,b\}\subseteq\mathcal{L},a\neq b} \mathrm{A}^\star_\mathrm{W}(\mathrm{I}_{ab}) \leq \sum_{\{a,b\}\subseteq\mathcal{L},a\neq b} (c\,\mathrm{B}^\star_\mathrm{W}(\mathrm{I}_{ab}) + b_1) \\ &= c \sum_{\{a,b\}\subseteq\mathcal{L},a\neq b} \mathrm{B}^\star_\mathrm{W}(\mathrm{I}_{ab}) + \sum_{\{a,b\}\subseteq\mathcal{L},a\neq b} b_1 = c\,\mathrm{B}^\star_\mathrm{W}(\mathrm{I}) + \binom{\ell}{2} b_1.\end{aligned}$$

Hence, we have the result in the partial cost model. Now assume A and B are cost independent and $c \geq 1$. It is clear that for a cost independent algorithm A, the cost in the partial and the full cost model are related as $\mathrm{A}_\mathrm{W}(\mathrm{I}) = \mathrm{A}^\star_\mathrm{W}(\mathrm{I}) + |\,\mathrm{I}\,|$. Hence, $\mathrm{A}^\star_\mathrm{W}(\mathrm{I}) \leq c\,\mathrm{B}^\star_\mathrm{W}(\mathrm{I}) + b$ implies that $\mathrm{A}_\mathrm{W}(\mathrm{I}) \leq c\,\mathrm{B}_\mathrm{W}(\mathrm{I}) + b$ and the result follows. □

It follows from the above that we can use list factoring to separate online algorithms, and an upper bound on the relative worst order ratio on lists of size two carries over to lists of any size. In particular, using $c = 1$, if algorithm A is never worse than algorithm B in the partial cost model, except for an additive constant, then this also applies to the full cost model. However, as it is also the case for competitive analysis, the list factoring technique cannot be used to prove lower bounds, i.e., even if algorithm A is at least a factor $c > 1$ worse than algorithm B in the partial cost model (except for an additive constant), this may not be the case in the full cost model.

For *randomized algorithms*, the worst ordering is defined in terms of the algorithm's expected cost when run on the sequence. In this case, a randomized algorithm is said to have either of the two properties if for all settings of the random choices made by the algorithm (a deterministic execution of the algorithm), the property holds. With this definition, it is clear that the list factoring technique can also be applied to randomized algorithms.

In the following, we repeatedly use the fact that MTF, FC, and TS have the pairwise property and are cost independent [10].

5 Worst Orderings

Intuition suggests that one can obtain a worst ordering of any sequence for most online list accessing algorithms by considering the request sequence as a multiset of items and always requesting the item from the multiset which currently is farthest back in the list.

Formally, for any deterministic online list accessing algorithm A and any request sequence I, we inductively define the *FB ordering* (Farthest Back ordering) of I as follows. Let S^0 be the multiset of all items requested in I. Let S^{i-1} be S^0 with the first $i-1$ items in the FB ordering removed. The ith item in the FB ordering of I with respect to A, $\mathrm{FB}_\mathrm{A}(\mathrm{I})_i$, is the item in S^{i-1} which currently is farthest back in the list after A has processed the first $i-1$ requests of $\mathrm{FB}_\mathrm{A}(\mathrm{I})$. In addition, we say that A has the *FB property* if for any request sequence the FB ordering of that sequence is a worst ordering with respect to A.

When the algorithm in question is obvious, we drop it from the notation and write FB(I). Note that for any deterministic algorithm and request sequence, the FB ordering of this input sequence is uniquely determined.

Observe that TRANS does not have the FB property as the following example illustrates. Consider the request sequence $\mathrm{I} = \langle a, b, c, c\rangle$ with the initial list $\mathcal{L} = (a, b, c)$. In this case, we have $\mathrm{FB}(\mathrm{I}) = \langle c, b, c, a\rangle$ with $\mathrm{TRANS}(\mathrm{FB}(\mathrm{I})) = 10$. However, on the ordering $\mathrm{I}' = \langle c, c, b, a\rangle$, TRANS incurs a cost of 11. Hence, FB(I) is not a worst ordering for TRANS.

The other deterministic algorithms considered in this chapter do have the FB property.

In the following three lemmas, the overall proof idea is the same when we want to show that some algorithm A has the FB ordering. We give it here to avoid repetition: Consider any request sequence and let I be a worst ordering of this sequence with respect to A. We gradually reorder this sequence into the FB ordering, maintaining at least the same cost, thereby proving the result. At each step we increase the length of the FB ordered prefix by at least one request. Hence, the process terminates after a number of steps bounded by the length of the request sequence.

Lemma 2 MTF *has the* FB *property.*

Proof Following the outline above, consider the first request I_i in I, which differs from the FB ordering of I, i.e., $\mathrm{I}_i = a \neq b = \mathrm{FB}(\mathrm{I})_i$. Let I_j be the first request to b in I after I_i. Such a request to b must exist since FB(I) is a permutation of I and the two sequences were identical up to I_i. If b is also requested later than I_j in I, let I_k be this next request. Otherwise, let I_k denote the last request in I.

Reorder I into I′ by moving $\mathrm{I}_j = b$ just in front of $\mathrm{I}_i = a$, i.e.,

$$\begin{aligned} \mathrm{I} &= \langle \cdots, \mathrm{I}_{i-1}, \quad a, \mathrm{I}_{i+1}, \cdots, \mathrm{I}_{j-1}, b, \mathrm{I}_{j+1}, \cdots \rangle, \text{ and} \\ \mathrm{I}' &= \langle \cdots, \mathrm{I}_{i-1}, b, a, \mathrm{I}_{i+1}, \cdots, \mathrm{I}_{j-1}, \quad \mathrm{I}_{j+1}, \cdots \rangle. \end{aligned}$$

First note that since MTF has the pairwise property, moving b in the request sequence only affects b's position in the list, the relative order of all other items remaining the same. Hence, if b is accessed more than once after I_j in I, then after the second access to b (at I_k), the list is ordered the same for I and I′. Consequently, we only need to consider items requested in the subsequence $\langle \mathrm{I}_i, \ldots, \mathrm{I}_k \rangle$ to prove that the cost of I′ is at least the cost of I. Again, since MTF has the pairwise property, for each of these items, we only need to consider the number of negative comparisons between b and the item accessed in both sequences.

Let d be any item requested in $\langle \mathrm{I}_i, \ldots, \mathrm{I}_k \rangle$, $d \neq b$. In the following, the pair (n, m) denotes that in I, d is requested n times in $\langle \mathrm{I}_i, \ldots, \mathrm{I}_{j-1} \rangle$ and m times in $\langle \mathrm{I}_{j+1}, \ldots, \mathrm{I}_k \rangle$. We now have several cases depending on n and m.

First, consider the pair $(0, m)$, $m \geq 0$. Since d is not accessed in the sequence $\langle \mathrm{I}_i, \ldots, \mathrm{I}_{j-1} \rangle$, the relative order of b and d in the list is the same whether they are requested in $\langle \mathrm{I}_i, \ldots, \mathrm{I}_k \rangle$ or $\langle \mathrm{I}'_i, \ldots, \mathrm{I}'_k \rangle$. Consequently, we have the same number of negative comparisons between b and d in the two sequences.

Next, consider the pair $(n, 0)$, $n > 0$. For both $\langle \mathrm{I}_i, \ldots, \mathrm{I}_k \rangle$ and $\langle \mathrm{I}'_i, \ldots, \mathrm{I}'_k \rangle$, we have a negative comparison with d at the first access to b, since b is at the end of the list. For I, there are no negative comparisons at the n accesses to d. If I_k is an access to b, this does not give rise to any negative comparisons, since, with $m = 0$, there are no accesses to d after b is moved to the front by request I_j. For I′, since b is moved to the front of the list at its first access, there is one negative comparison at the first of the n accesses to d. Also, if I_k is an access to b, there is always one negative comparison at this access. Hence, in this case the number of negative comparisons always increase.

Now consider the pair (n, m), $n > 0, m > 0$. Again, for both $\langle \mathrm{I}_i, \ldots, \mathrm{I}_k \rangle$ and $\langle \mathrm{I}'_i, \ldots, \mathrm{I}'_k \rangle$, we have one negative comparison at the first access to b. For I, since d is before b in the list just after I_{i-1}, there are no negative comparisons at the first n accesses to d. There is one negative comparison at the first of the m accesses to d, and one at the access I_k if it is an access to b. For I', there is one negative comparison at the first of the $n+m$ accesses to d, and a negative comparison at the access I_k if it is an access to b. Hence, in this case the number of negative comparisons is the same in I' as in I.

In all cases, the cost for MTF when processing I' with respect to b and d is at least the same as the cost when processing I. □

Lemma 3 TS *has the* FB *property.*

Proof We follow the outline given immediately before Lemma 2. Observe that for TS and any input sequence, the ordering of the items in the list at any point in time only depends on the initial ordering of the items and the last two accesses to each item. By this observation and the definition of TS, it follows by induction that the FB ordering of a request sequence repeatedly accesses the same item twice in a row (possibly except for the last access to any item), i.e., the item farthest back in the list. Note that the second access moves the item to the front of the list.

Divide FB(I) into phases corresponding to these pairs, i.e., a phase has length one or two.

Consider the first request I_i in I which differs from the FB ordering, i.e., $\mathrm{FB(I)}_i = a \neq \mathrm{I}_i$. Hence, a is the item farthest back of the items with remaining requests.

First, assume that $\mathrm{FB(I)}_i$ is the only request in its phase, i.e., this is the last access to a in FB(I). By the observation above, a does not change its position in the list. It follows that the cost of this access is the same, independently of when it is made, i.e., we maintain the same cost by moving the one remaining request for a in I forward to just before I_i and as a consequence increase the prefix which is identical with the FB ordering.

Now, assume that $\mathrm{FB(I)}_i$ is the first of two requests to a in its phase. Further, assume that at least two other requests to a remains in FB(I) after this phase. We reorder I in the following way (a_i, $1 \leq i \leq 4$, are the first four of the remaining requests to a).

$$\mathrm{I} = \langle \cdots, \overbrace{\mathrm{I}_i, \cdots}^{A}, a_1, \overbrace{\cdots}^{B}, a_2, \overbrace{\cdots}^{C}, a_3, \overbrace{\cdots}^{D}, a_4, \cdots \rangle, \text{ and}$$
$$\mathrm{I}' = \langle \cdots, a_1, a_2, \overbrace{\mathrm{I}_i, \cdots}^{A}, \overbrace{\cdots}^{B}, \overbrace{\cdots}^{C}, a_3, \overbrace{\cdots}^{D}, a_4, \cdots \rangle.$$

We need to show that the cost for TS to serve I' is at least as high as the cost for serving I. By the observation about TS, the behavior of TS after a_4 is the same for both sequences. Consider any item $d \neq a$ which is requested in A, B, C, or D. Since TS has the pairwise property, we only need to show that the number of negative comparisons between d and a has not decreased. Again, it follows from the observation that we can assume d is requested $0, 1$, or 2 times in each subsequence.

Recall that a does not change its position as the last item in the list when accessed at a_1 (in both sequences). Hence, for both sequences, there are negative comparisons between a and d at the accesses a_1 and a_2. In I, there are no negative comparisons in A and B.

First assume that d is requested two or more times in total in A, B, and C. In I′, we have two negative comparisons in total in A, B, and C and one at each of a_3 and a_4 for a total of four negative comparison between a and d. Now consider I. If d occurs two or more times in C, we have two negative comparisons in C and one at both a_3 and a_4 (no negative comparisons in D since a is not moved in front of d at a_3). If d occurs only once in C, we have at most one negative comparison in C, one at a_3, at most one in D, and at most one negative comparison at a_4. Finally, if d does not occur in C, we have at most one negative comparison at a_3, two in D, and one at a_4. In all cases, we have at most four negative comparisons. Hence, the cost for TS to serve I′ is at least as high as the cost for serving I.

Next, assume that d is requested once in total in A, B, and C. In I′, we have one negative comparison in total in A, B, and C, and if d occurs in D, we have one negative comparison in D and one at a_4. Now consider I. As noted above, if the access to d occurs in either A or B, then we have no negative comparisons in total in A, B, and C. If the access to d occurs in C, then we have one negative comparison at that access. If d occurs in D, we have one negative comparison in D and one at a_4. In all cases, we have at least the same number of negative comparisons in I′ as in I.

As the last case, assume that d does not occur in A, B, and C. It is clear that from the perspective of a and d, the two sequences I and I′ are identical. Hence, they have the same number of negative comparisons between a and d.

Finally, observe that the arguments above still hold in the case where there are one or no further requests for a after a_2, i.e., a_4 or both a_4 and a_3 do not exist. Similarly, if $\mathrm{FB}(\mathrm{I})_i$ is not the first but the second of the two requests for a in its subphase, the arguments still hold (this case corresponds to A being the empty sequence). □

Lemma 4 FC *has the* FB *property.*

Proof Following the outline given immediately before Lemma 2, consider the first item in I which differs from the FB ordering of I, i.e., $\mathrm{I}_i \neq \mathrm{FB}(\mathrm{I})_i = b$. Let I_j be the first request to b in I after I_i. Such a request to b must exist since FB(I) is a permutation of I and the two sequences were identical up to I_i. Reorder I into I′ by swapping I_j with the preceding item, i.e., with the item at position $\mathrm{I}_{j-1} = c$.

$$\mathrm{I} = \langle \cdots, \mathrm{I}_{j-2}, c, b, \mathrm{I}_{j+1}, \cdots \rangle, \text{ and}$$
$$\mathrm{I}' = \langle \cdots, \mathrm{I}_{j-2}, b, c, \mathrm{I}_{j+1}, \cdots \rangle.$$

Below, we show that the cost incurred by FC in serving I′ is at least the cost incurred when serving I. If we can accomplish this, then we can use this technique repeatedly to move b all the way from its location as the jth request in the input sequence closer to the front as the ith request in the input sequence without reducing the overall cost. This would conclude the proof.

We now prove that I′ incurs at least the same cost as I. Since FC has the pairwise property, we only need to consider the relative positions of b and c and the number of negative comparisons between the two.

We have three cases.

First, assume that the frequency of c just after I_{j-2} is lower than the frequency of b. The frequency of c just before I_i was also lower than the frequency of b at that point. Hence, by the FC policy c was further back than b in the list just before I_i. Thus, the next request in the FB ordering would not have been to b, and we have reached a contradiction with the assumption that b is the ith request in the FB ordering.

Next, assume that the frequency of c just after I_{j-2} is equal to the frequency of b. Since b is further back in the list than c, there is one extra negative comparison in I′ in comparison with I up until just before I_{j+1}. However, the relative ordering of b and c in the list is now reversed going from I to I′, which may later cause one fewer negative comparison in I′ in comparison with I. Overall, the number of negative comparisons have not decreased.

Finally, assume that the frequency of c just after I_{j-2} is higher than the frequency of b. In this case, c stays in front of b in the list for both sequences. Hence, the cost for FC to serve both sequences is exactly the same. □

When applying the list factoring technique in the next section, we need the following lemma and the corollary immediately implied by it.

Lemma 5 *If a deterministic algorithm has the* FB *property and the pairwise property, it also has the worst order projection property.*

Proof The following is stated in [10, Lemma 1.1]: An algorithm A satisfies the pairwise property if and only if for every request sequence I, when A serves I, the relative order of every two elements a and b in the list is the same as their relative order when A serves I_{ab}.

Now, consider an algorithm A with both the FB property and the pairwise property. Let I be any request sequence in FB order, and let a and b be two different items in I. By the above, the relative order of these two items is the same when A serves I as when it serves I_{ab}, hence I_{ab} is also an FB ordering. As A has the FB property, I_{ab} is a worst possible request sequence. Since this is true for any a and b, A also has the worst order projection property. □

Corollary 1 MTF*,* FC*, and* TS *have the worst order projection property.*

6 Algorithm Comparisons

We now have the tools necessary to compare the online list accessing algorithms.

6.1 Deterministic Algorithms

When comparing MTF to FC and MTF to TS below, we apply the list factoring technique introduced in Section 4 since all the considered algorithms have the pairwise, FB, and worst order projection properties. The proofs are carried out in the partial cost model, but since, for any cost independent algorithm A, $A^\star(I) - A(I) = |I|$, they immediately carry over to the full cost model.

Theorem 1 *In both the partial and full cost model, the algorithms* MTF *and* FC *perform identically according to relative worst order analysis.*

Proof Following the outline from above, consider any request sequence I and any pair $\{a, b\} \subseteq \mathcal{L}$, $a \neq b$. Assume without loss of generality that the initial list has a in front of b, i.e., $\mathcal{L}_{ab} = (a, b)$.

Now, the FB ordering of I_{ab} for MTF is of the form $\langle (b, a)^m \rangle$ with a possible tail of repeated requests to either a or b, whichever is requested the most in I. The FB ordering for FC is of the form $\langle (b, a, a, b)^{\lfloor \frac{m}{2} \rfloor} \rangle$ with a possible tail of repeated requests to either a or b, whichever is requested the most in I. Observe that if m is not divisible by two, there is an extra request to either a or b. However, such a request only contributes a constant extra cost which we can ignore. It now follows that the cost for FC on its worst permutation (the FB ordering) is the same as the cost for MTF on its worst permutation (the FB ordering), except for a possible additive constant. □

Theorem 2 *In both the partial and full cost model, the algorithms* MTF *and* TS *perform identically according to relative worst order analysis.*

Proof Following the outline from above, consider any request sequence I and any pair $\{a, b\} \subseteq \mathcal{L}$, $a \neq b$. Assume without loss of generality that the initial list projected onto a and b has a at the front, i.e., $\mathcal{L}_{ab} = (a, b)$.

The FB ordering of I_{ab} for TS is of the form $\langle (b, b, a, a)^{\lfloor \frac{m}{2} \rfloor} \rangle$. The remaining arguments are exactly the same as in the proof of Theorem 1. □

Combining the previous two theorems and using the fact that the relative worst order ratio is a transitive measure, we arrive at the following corollary.

Corollary 2 *In both the partial and full cost model, the algorithms* MTF, TS, *and* FC *perform identically according to relative worst order analysis.*

We now show that TRANS cannot be better than any of MTF, TS, and FC according to relative worst order analysis.

Lemma 6 *In both the partial and full cost model, there exists an additive constant b such that for any request sequence, the cost of* MTF *is at most b larger than the cost of* TRANS *on their worst orderings.*

Proof The proof is carried in the full cost model, but in the same way as in the outline from above, the result immediately carries over to the partial cost model.

Consider any request sequence I. Reorder I into an FB ordering with respect to MTF, I_{MTF}, and recursively divide it into phases as follows. Each phase starts where the previous phase ends. Let n_d be the number of distinct items in the remaining part of the sequence (the part which has not yet been divided into phases). The next phase contains the next n_d requests in the sequence, and these are all distinct. This is because I_{MTF} is an FB ordering with respect to MTF and MTF moves items to the front at each request. Thus, after having moved one item to the front, the other n_d-1 items must be requested before the same item is again farthest back. Thus, the requests in I_{MTF} appear cyclically, and therefore the relative order of the items accessed in a phase is the same immediately before and after the phase.

Now group the phases into super phases, where each super phase is a maximal sequence of consecutive phases under the restriction that the phases request the same number of distinct items. Hence, the accesses in each phase of a super phase are to the same set of items. Let r_i denote the number of phases in the ith super phase, and let n_i denote the number of distinct items accessed in each phase of the ith super phase. We have numbered the super phases starting from one. However, for mathematical convenience, we define $n_0 = \ell$, recalling that ℓ is the length of the list $\mathcal{L}$.

Observe that the number of distinct items in the super phases are decreasing and the number of distinct items in the first phase is at most ℓ. Hence, there are at most ℓ super phases.

For MTF, the cost of the ith super phase can now be calculated as follows. The first phase of the ith super phase costs at most $n_{i-1}n_i$, since the n_i accesses in the worst case is for items at index n_{i-1} in the list (by the MTF policy). The remaining $r_i - 1$ phases in the super phase cost n_i^2 each. Hence, the total cost for MTF of the ith super phase is $n_i(n_{i-1} - n_i) + r_i n_i^2$. The only term we are interested in is $r_i n_i^2$, since the remaining terms, over all super phases, can be bounded by a constant depending only on ℓ. Thus, this constant is independent of the length of the request sequence.

Hence, we need to show that we can find an ordering making TRANS incur a cost of at least $r_i n_i^2$, up to an additive constant.

We ignore super phases with $r_i \leq 2\ell$. The cost for MTF on such phases is at most $r_i n_i \ell \leq 2\ell^3$. Since there are at most ℓ phases, the total cost incurred by MTF on these super phases is only a constant dependent on ℓ.

Now, assume that the ith super phase has $r_i > 2\ell$. We reorder the requests for TRANS. First, we access each of the n_i items ℓ times, which moves the n_i items to the first n_i positions in the list. Assume that the first n_i items in the list after this are $(a_1, a_2, \ldots, a_{n_i})$ where a_1 is at the front of the list.

First, if n_i is one, simply repeatedly access the item, giving a total cost of at least $r_i = r_i n_i^2$.

Next, if n_i is even, let $r_i' = r_i - \ell - n_i > 0$. Access a_{n_i} and $a_{n_{i-1}}$ alternately r_i' times with a cost of n_i for each access. Subsequently, access a_{n_i} n_i times, and then $a_{n_{i-1}}$ n_i times, thereby moving them to the front of the list. Repeat this process for the remaining $n_i - 2$ items in groups of two. The total cost is at least $(\ell + r_i' n_i + n_i)\, n_i = \ell n_i + r_i n_i^2 - \ell n_i^2 - n_i^3 + n_i^2$.

Finally, if n_i is odd and at least three, we do as in the previous case, except when we are down to the last three items, a_1, a_2, and a_3 at positions n_i, n_{i-1}, and n_{i-2}. Let $r_i'' = \left\lfloor \frac{r_i-\ell-2}{2} \right\rfloor$ and request

$$\langle (a_1, a_2)^{r_i''}, a_1, a_1, (a_2, a_3)^{r_i''}, a_2, a_2, (a_3, a_1)^{r_i''}, a_3, a_3 \rangle.$$

All requests have a cost of n_i except for the last of the double accesses to a_1, a_2, and a_3 each with a cost of $n_i - 1$. In total, the three items are each accessed $\ell + 2r_i'' + 2 \in \{r_i - 1, r_i\}$ times. If this is $r_i - 1$, request the items $\langle a_1, a_2, a_3 \rangle$, each at a cost of n_i. In total, the cost per item is $r_i n_i - 1$.

In both of the above two cases, we can ignore all terms, except for terms involving r_i. Hence, the total cost for TRANS in this super phase is at least $r_i n_i^2$, except for a constant dependent only on ℓ, i.e., the difference in cost for MTF and TRANS is bounded by a constant only dependent on ℓ. □

On the other hand, TRANS can be much worse than MTF, FC, and TS under relative worst order analysis.

Theorem 3 *In both the partial and full cost model,* MTF *performs better than* TRANS *with a relative worst order ratio of* $\mathrm{WR}_{\mathrm{TRANS,MTF}} \geq \frac{\ell}{2}$ *and* $\mathrm{WR}_{\mathrm{TRANS,MTF}} \geq \ell - 1$*, respectively.*

Proof Lemma 6 shows that TRANS cannot be better than MTF according to relative worst order analysis. Assume that the initial list is $\mathcal{L} = (a_1, a_2, \ldots, a_\ell)$ and consider the request sequence $\mathrm{I} = \langle (a_\ell, a_{\ell-1})^m \rangle$.

In the partial cost model, it is clear that MTF incurs a cost of $2(\ell - 1) + 2(m-1)$ on its worst permutation of I. On the other hand, TRANS leaves the two items at the end of the list and incurs a cost of $2m(\ell - 1)$. For m approaching infinity, the ratio approaches $\ell - 1$.

In the full cost model, MTF has a cost of $2\ell + 4(m-1)$, whereas TRANS has a cost of $2m\ell$. For m approaching infinity, the ratio approaches $\frac{\ell}{2}$. □

6.2 Randomized Algorithms

In this section, to make the proofs more readable, we use the partial cost model, except for the final algorithm comparison which is in both cost models.

Lemma 7 *For integers $n \geq 1$ and $m \geq 2$ and a request sequence $\mathrm{I} = \langle (b, a^m)^n \rangle$ with initial list $\mathcal{L} = (a, b)$, the expected cost of* BIT *for a single repetition of $\langle b, a^m \rangle$ is $\frac{7}{4}$, and* I *is its own worst permutation with respect to* BIT.

Proof For each access to b, at most the next two accesses to a contribute to the expected cost of BIT. It follows by induction that after each repetition of $\langle b, a^m \rangle$, a is at the front of the list for BIT. Hence, the expected cost of the prefix $\langle b, a \rangle$ of the next repetition is $\frac{3}{2}$, and after that a is at the front of BIT's list with probability $\frac{3}{4}$. Thus, the expected cost of the following access to a is $\frac{1}{4}$, after which a is at the front of BIT's list with probability 1, and the remaining accesses to a in the current repetition do not cost anything. Hence, for any m, the total expected cost of a single repetition is $\frac{7}{4}$ for BIT. It is clear that I is its own worst permutation for BIT. □

The following three lemmas establish various properties regarding RMTF's behavior on sequences of the form $\langle (b, a^m)^n \rangle$. Together, these results will enable us to prove that there exists a sequence where RMTF has a better expected cost than BIT.

Lemma 8 *The following holds concerning* RMTF*'s behavior on sequences of the form* $\langle b, a^m \rangle$*, for* $m \geq 1$*, and for repetitions of such sequences of the form* $\langle (b, a^m)^n \rangle$*, for* $n \geq 1$*.*

- *If* a *is in the front of the list with probability* p*, then after serving* $\langle b, a^m \rangle$*,* a *is in the front of the list with probability* $1 - \frac{2-p}{2^{m+1}}$ *and the expected cost of serving this sequence is* $c_m(p) = 2 - \frac{2-p}{2^m}$*.*
- *For* n *going towards infinity, the probability of* a *being at the front of the list after each repetition of* $\langle b, a^m \rangle$ *approaches* $p_m = 1 - \frac{1}{2^{m+1}-1}$ *and the expected cost of serving each repetition approaches* $\frac{2^{m+2}-4}{2^{m+1}-1}$*.*

Proof Assume that a is at the front of the list with probability p and consider a sequence $\langle b, a^m \rangle$.

After the access to b, a is not at the front with probability $1 - \frac{p}{2}$. In this case, the up to m requests to a while it is not at the front can be described by a truncated geometric distribution [10, Lemma 4.1] with an expected number of $2\left(1 - \frac{1}{2^m}\right)$. Hence, the total cost of this sequence is

$$c_m(p) = p + \left(1 - \frac{p}{2}\right) 2 \left(1 - \frac{1}{2^m}\right) = 2 - \frac{2-p}{2^m}.$$

The probability of a being at the front of the list after the repetition is then

$$1 - \frac{1 - \frac{p}{2}}{2^m} = 1 - \frac{2-p}{2^{m+1}}.$$

Now, consider the sequence $\langle (b, a^m)^n \rangle$ for n approaching infinity. The probability of a being at the front of the list after each repetition of $\langle b, a^m \rangle$ approaches p_m, where

$$p_m = 1 - \frac{2 - p_m}{2^{m+1}} \Rightarrow p_m = 1 - \frac{1}{2^{m+1} - 1}.$$

Hence, the cost of a repetition approaches

$$c_m(p_m) = 2 - \frac{2 - p_m}{2^m} = 2 - \frac{1}{2^m} - \frac{1}{2^m(2^{m+1} - 1)} = \frac{2^{m+2} - 4}{2^{m+1} - 1}.$$

□

Table 1 lists the results from Lemma 8 for small m and bounds the values for larger m.

m	$c_m(p)$	p after subseq.	p_m	$c_m(p_m)$
0	p	$\frac{p}{2}$	0	0
1	$1+\frac{p}{2}$	$\frac{1}{2}+\frac{p}{4}$	$\frac{2}{3}$	$\frac{4}{3}$
2	$\frac{3}{2}+\frac{p}{4}$	$\frac{3}{4}+\frac{p}{8}$	$\frac{14}{15}$	$\frac{12}{7}$
3	$\frac{7}{4}+\frac{p}{8}$	$\frac{7}{8}+\frac{p}{16}$	$\frac{30}{31}$	$\frac{28}{15}$
≥ 4	≤ 2	≤ 1	≤ 1	≤ 2

Table 1 Consider RMTF serving $\langle (b, a^m)^n \rangle$, i.e., n repetitions of the subsequence $\langle b, a^m \rangle$. Assume that before serving a subsequence, the probability of a being at the front of the list is p. For various m, the table lists the expected cost $c_m(p)$ of serving the subsequence, the probability of a being at the front of the list after serving the subsequence, the limit value p_m of the probability of a being at the front of the list for n going towards infinity, and the expected cost $c_m(p_m)$ of serving a subsequence for n going towards infinity.

Lemma 9 *Consider a sequence $\langle (b, a, a)^n \rangle$ for some positive integer n with the initial list $\mathcal{L} = (a, b)$. The expected cost of* RMTF *serving this sequence if a is initially at the front of the list with any probability p is at most $\frac{2}{49}$ larger than the expected cost of* RMTF *serving this sequence if a is initially at the front of the list with probability $c_2(p_2)$.*

Proof As a worst case assumption, we assume that the length of the sequence is infinite and that $p = 1$.

Starting with a probability of 1, the probability of a being at the front of the list just before RMTF starts to serve subsequence j, $j \geq 0$, is,

$$\frac{3}{4} + \frac{1}{8}\cdot\frac{3}{4} + \frac{1}{8^2}\cdot\frac{3}{4} + \cdots + \frac{1}{8^{j-1}}\cdot\frac{3}{4} + \frac{1}{8^j}$$
$$= \frac{3}{4}\left(\sum_{i=0}^{j-1}\frac{1}{8^i}\right) + \frac{1}{8^j} = \frac{3}{4}\cdot\frac{1-\frac{1}{8^j}}{1-\frac{1}{8}} + \frac{1}{8^j} = \frac{6}{7} + \frac{1}{7}\cdot\frac{1}{8^j}.$$

Hence, the contribution of subsequence j to the extra cost is

$$c_2\left(\frac{6}{7} + \frac{1}{7}\cdot\frac{1}{8^j}\right) - c_2(p_2) = \frac{3}{2} + \frac{6}{28} + \frac{1}{28}\cdot\frac{1}{8^j} - \frac{12}{7} = \frac{1}{28}\cdot\frac{1}{8^j}.$$

Summing all the contributions, we get the extra cost of the sequence beginning with $p = 1$ in comparison with starting with probability $c_2(p_2)$ expressed as

$$\sum_{j=0}^{\infty}\frac{1}{28}\cdot\frac{1}{8^j} = \frac{1}{28}\cdot\frac{1}{1-\frac{1}{8}} = \frac{2}{49}.$$

□

Lemma 10 *Consider the request sequence* I $= \langle (b, a, a)^n \rangle$ *for some positive integer n with the initial list $\mathcal{L} = (a, b)$. The sequence* I *is a worst ordering with respect to* RMTF*.*

Proof Let I′ be any other ordering of the requests in I. We prove that the expected cost of serving I is at least as high as the expected cost of serving I′.

Since a is initially at the front of the list, we assume without loss of generality that the first request in I′ is for b.

We divide the sequences I and I′ up into phases. Each phase starts with a b and continues until just before the next b (or to the end of the sequence if there are no more b's). We let m denote the number of a's in a phase. The proof idea is to match phases in I′ for values of $m \neq 2$ up against a number of phases in I (which all have $m = 2$), such that the number of a's and b's correspond.

We let *stable state* refer to the situation we are approaching for growing n when serving I, i.e., the situation where a is at the front of the list with probability $c_2(p_2)$.

As a worst case assumption, we assume that whenever we start serving phases with $m \neq 2$ in I′, a is at the front of the list with probability 1.

Before we start matching up phases, we make the following observation which enables us to reduce the number of cases to be considered later. Consider the case where a phase in I′ is followed by a sequence of phases with $m = 2$. If we start processing such a following sequence of phases with $m = 2$ in a situation where a is at the front of the list with probability $p > \frac{6}{7}$, then the cost of the sequence will be higher than if we started in the stable state. However, by Lemma 9, we have upper bounded the extra cost by $\frac{2}{49}$. Now, by Table 1, only phases with $m \geq 3$ can end with $p > \frac{6}{7}$. Hence, in the following matching of phases below, we add a contribution of $\frac{2}{49}$ to the cost whenever we match a phase with $m \geq 3$. Then we do not need to be concerned with how many phases with $m = 2$ follows a phase with $m \neq 2$.

Consider the phases in I′. We repeatedly apply the following matching of phases until there are no phases or only phases with $m = 2$ are left, which we have accounted for by adding a contribution of $\frac{2}{49}$ to the possibly preceding phase.

- If there is an unmatched phase with $m = 3$ and an unmatched phase with $m = 1$ left in I′, then they correspond to exactly two phases with $m = 2$. An upper bound on the cost for RMTF when serving the requests in I′ is then
$$\left(\frac{7}{4} + \frac{1}{8}\right) + \left(1 + \frac{1}{2}\right) + \frac{2}{49},$$
which is strictly less than the cost of the corresponding two phases in I when RMTF is in the stable state, $2 \cdot \frac{12}{7}$.
- If there are two unmatched phases with $m = 3$ and an unmatched phase with $m = 0$ left in I′, then they correspond to exactly three phases with $m = 2$. An upper bound on the cost for RMTF when serving the requests in I′ is then
$$2\left(\frac{7}{4} + \frac{1}{8}\right) + 1 + 2 \cdot \frac{2}{49},$$
which is strictly less than the cost of the corresponding three phases in I when RMTF is in the stable state, $3 \cdot \frac{12}{7}$.

- If there are still phases with $m = 3$ left in I′ after applying the above two cases (repeatedly), then it follows that there must exist at least one phase with an odd $m > 3$ and a number of phases with $m = 0$ such that the resulting number of a's in the phases is two times the number of phases. Let x denote the value of m in the unmatched phase in I′ with the smallest value of m under the restriction that m is odd and $m \geq 5$. Then there are at least $\frac{3+x}{2} - 2 = \frac{x-1}{2}$ phases with $m = 0$. These phases in I′ correspond to $\frac{3+x}{2}$ phases with $m = 2$ in I. An upper bound on the cost for RMTF when serving the requests in I′ is then

$$\left(\frac{7}{4} + \frac{1}{8}\right) + 2 + \frac{x-1}{2} \cdot 1 + 2 \cdot \frac{2}{49} = \frac{x}{2} + \frac{1355}{392},$$

  which is strictly less than the cost of the corresponding phases in I when RMTF is in the stable state, $\frac{3+x}{2} \cdot \frac{12}{7} = \frac{6}{7}x + \frac{18}{7}$, for $x \geq 5$.
- If there is an unmatched phase with $m = 4$ and two unmatched phases with $m = 1$ left in I′, then they correspond to three phases with $m = 2$. An upper bound on the cost for RMTF when serving the requests in I′ is then

$$2 + 2\left(1 + \frac{1}{2}\right) + \frac{2}{49},$$

  which is strictly less than the cost of the corresponding three phases in I when RMTF is in the stable state, $3 \cdot \frac{12}{7}$.
- If there is an unmatched phase with $m = 4$ and an unmatched phase with $m = 0$ left in I′, then they correspond to two phases with $m = 2$. An upper bound on the cost for RMTF when serving the requests in I′ is then

$$2 + 1 + \frac{2}{49},$$

  which is strictly less than the cost of the corresponding two phases in I when RMTF is in the stable state, $2 \cdot \frac{12}{7}$.
- Finally, if there is an unmatched phase with $m > 4$, let x denote the value of m in the unmatched phase, and set $y = \lceil \frac{x}{2} \rceil \geq 3$. In I′, there must be $y - 1$ phases with $m \leq 1$. As a worst case assumption, we assume that all such phases have $m = 1$. Now, an upper bound on the cost for RMTF when serving the requests in I′ is then

$$2 + (y-1)\left(1 + \frac{1}{2}\right) + \frac{2}{49},$$

  which is strictly less than the cost of the corresponding y phases in I when RMTF is in the stable state, $y\frac{12}{7}$, for $y \geq 3$.

Observe that the above covers all cases. Hence, the ordering in I is indeed a worst ordering for RMTF. □

Combining the lemmas above, we can now prove that there exists a sequence where RMTF has a better expected cost than BIT.

Theorem 4 *In both the partial and full cost model, there exists a request sequence* I *such that the expected cost for* RMTF *on its worst permutation of* I *is a constant factor of* $c < 1$ *less than the expected cost for* BIT *on its worst permutation.*

Proof Consider the request sequence $\mathrm{I} = \langle (b, a, a)^n \rangle$ for some positive integer n with the initial list $\mathcal{L} = (a, b)$. First, consider this in the partial cost model.

For BIT, by Lemma 7, the expected cost of each repetition of $\langle b, a, a \rangle$ is $\frac{7}{4}$, and by Lemma 10, I is a worst permutation with respect to RMTF. Thus, we just need to argue that the expected cost of RMTF for serving I is strictly smaller than the cost of BIT.

For the first repetition, a is at the front, so the expected cost for RMTF is $c_2(1) = \frac{7}{4}$. For all subsequent repetitions, the probability p of a being at the front of the list is strictly smaller than one, and the expected cost of serving the repetition is then $c_2(p) < \frac{7}{4}$ (see Table 1). The expected cost of a repetition approaches $c_2(p_2) = \frac{12}{7}$ for n approaching infinity. This is strictly smaller than $\frac{7}{4}$, which was the cost of BIT.

In the full cost model, for both algorithms, the cost of each repetition increases by three, i.e., RMTF is still a constant factor better than BIT. □

Lemma 11 *In both the partial and full cost model, there exists a request sequence* I *such that the expected cost for* BIT *on its worst permutation of* I *is a constant factor of* $c < 1$ *less than the expected cost for* RMTF *on its worst permutation.*

Proof Consider the request sequence $\mathrm{I} = \langle (b, a, a, a)^n \rangle$ for some positive integer n with the initial list $\mathcal{L} = (a, b)$. First consider this in the partial cost model.

For BIT, by Lemma 7, the cost of each repetition of $\langle b, a, a, a \rangle$ is $\frac{7}{4}$, and I is its own worst permutation.

For RMTF, by Table 1, the cost of each repetition approaches $c_3(p_3) = \frac{28}{15}$ from above, which is strictly more than $\frac{7}{4}$.

In the full cost model, for both algorithms, the cost of each repetition increases by four, i.e., BIT is still a constant factor better than RMTF. □

An interesting observation is that the sequences used in the previous lemmas are both repetitions of the pattern $\langle b, a^m \rangle$ for the values of $m = 2$ and $m = 3$, respectively.

The previous two results, Theorem 4 and Lemma 11, imply the following:

Corollary 3 BIT *and* RMTF *are not comparable using relative worst order analysis.*

7 Open Problems

In order to apply the list factoring technique together with relative worst order analysis, both of the *pairwise property* and the *worst order projection property* must hold. By Lemma 5, the pairwise property together with the FB property implies the worst order projection property. Apart from this lemma, we have not been able to show a direct dependence between the two properties, i.e., does one follow directly from the other? On the other hand, we have not been able to exhibit an example for which one holds and the other does not.

Another interesting question is whether the list factoring technique can be used with performance measures other than competitive analysis and, as demonstrated here, relative worst order analysis.

Acknowledgments

The authors would like to thank Joan Boyar for initial discussions on the relationship between MTF and TRANS and the anonymous referees for constructive suggestions for improving the paper.

References

1. Albers, S.: Improved randomized on-line algorithms for the list update problem. SIAM Journal on Computing **27**(3), 682–693 (1998)
2. Albers, S., Lauer, S.: On list update with locality of reference. In: Proceedings of the 35th International Colloquium on Automata, Languages and Programming, pp. 96–107 (2008)
3. Albers, S., von Stengel, B., Werchner, R.: A combined BIT and TIMESTAMP algorithm for the list update problem. Information Processing Letters **56**, 135–139 (1995)
4. Albers, S., Westbrook, J.: Self-organizing data structures. In: A. Fiat, G.J. Woeginger (eds.) Online Algorithms — The State of the Art, *Lecture Notes in Computer Science*, vol. 1442, pp. 13–51. Springer (1998)
5. Angelopoulos, S., Dorrigiv, R., López-Ortiz, A.: On the separation and equivalence of paging strategies. In: 18th ACM-SIAM Symposium on Discrete Algorithms, pp. 229–237 (2007)
6. Angelopoulos, S., Dorrigiv, R., López-Ortiz, A.: List update with locality of reference. In: Proceedings of the 8th Latin American Symposium on Theoretical Informatics, pp. 399–410 (2008)
7. Bachrach, R., El-Yaniv, R.: Online list accessing algorithms and their applications: Recent empirical evidence. In: Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 53–62 (1997)
8. Ben-David, S., Borodin, A.: A New Measure for the Study of On-Line Algorithms. Algorithmica **11**, 73–91 (1994)
9. Bentley, J.L., McGeoch, C.C.: Amortized analyses of self-organizing sequential search heuristics. Communications of the ACM **28**, 404–411 (1985)
10. Borodin, A., El-Yaniv, R.: Online Computation and Competitive Analysis. Cambridge University Press (1998)
11. Boyar, J., Favrholdt, L.M.: The relative worst order ratio for on-line algorithms. ACM Transactions on Algorithms **3**(2), article 22, 24 pages (2007)
12. Boyar, J., Favrholdt, L.M., Larsen, K.S.: The relative worst-order ratio applied to paging. Journal of Computer and System Sciences **73**, 818–843 (2007)
13. Boyar, J., Irani, S., Larsen, K.S.: A Comparison of Performance Measures for Online Algorithms. In: Eleventh International Algorithms and Data Structures Symposium, *Lecture Notes in Computer Science*, vol. 5664, pp. 119–130. Springer (2009)
14. Boyar, J., Medvedev, P.: The relative worst order ratio applied to seat reservation. ACM Transactions on Algorithms **4**(4), article 48, 22 pages (2008)

15. Dorrigiv, R., Ehmsen, M.R., López-Ortiz, A.: Parameterized analysis of paging and list update algorithms. In: Proceedings of the 7th International Workshop on Approximation and Online Algorithms, *Lecture Notes in Computer Science*, vol. 5893, pp. 104–115. Springer (2010)
16. Dorrigiv, R., López-Ortiz, A.: A Survey of Performance Measures for On-line Algorithms. SIGACT News **36**(3), 67–81 (2005)
17. Epstein, L., Favrholdt, L.M., Kohrt, J.S.: Comparing online algorithms for bin packing problems. Journal of Scheduling Accepted for publication.
18. Epstein, L., Favrholdt, L.M., Kohrt, J.S.: Separating scheduling algorithms with the relative worst order ratio. Journal of Combinatorial Optimization **12**(4), 362–385 (2006)
19. Garefalakis, T.: A new family of randomized algorithms for list accessing. In: Proceedings of the 5th Annual European Symposium on Algorithms, pp. 200–216 (1997)
20. Graham, R.L.: Bounds on multiprocessing timing anomalies. SIAM Journal on Applied Mathematics **17**(2), 416–429 (1969)
21. Irani, S.: Two results on the list update problem. Information Processing Letters **38**(6), 301–306 (1991)
22. Karlin, A.R., Manasse, M.S., Rudolph, L., Sleator, D.D.: Competitive snoopy caching. Algorithmica **3**, 79–119 (1988)
23. Kenyon, C.: Best-fit bin-packing with random order. In: 7th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 359–364 (1996)
24. Koutsoupias, E., Papadimitriou, C.H.: Beyond Competitive Analysis. SIAM Journal on Computing **30**(1), 300–317 (2001)
25. Krumke, S.O., de Paepe, W.E., Rambau, J., Stougie, L.: Bincoloring. Theoretical Computer Science **407**(1–3), 231–241 (2008)
26. McCabe, J.: On serial files with relocatable records. Operations Research **13**(4), 609–618 (1965)
27. Reingold, N., Westbrook, J., Sleator, D.D.: Randomized competitive algorithms for the list update problem. Algorithmica **11**, 15–32 (1994)
28. Sleator, D.D., Tarjan, R.E.: Amortized efficiency of list update and paging rules. Communications of the ACM **28**(2), 202–208 (1985)
29. Teia, B.: A lower bound for randomized list update algorithms. Information Processing Letters **47**, 5–9 (1993)
30. Young, N.: The k-server dual and loose competitiveness for paging. Algorithmica **11**, 525–541 (1994)