ELSEVIER

Theoretical Computer Science 242 (2000) 313–325

Theoretical Computer Science

www.elsevier.com/locate/tcs

# On the competitiveness of the move-to-front rule [1]

Conrado Martínez *, Salvador Roura

*Universitat Politècnica de Catalunya, Departament Llenguages i Sistemes Informatics, Campus Nord, Jordi Girona, 1-3, E-08034 Barcelona, Spain*

Received July 1997; revised September 1998
Communicated by J. Díaz

**Abstract**

We consider the list access problem and show that one questionable assumption in the original cost model presented by Sleator and Tarjan (1985) and subsequent literature allowed for several competitiveness results of the move-to-front rule (MTF). We present an off-line algorithm for the list access problem and prove that, under a more realistic cost model, no on-line algorithm can be $c$-competitive for any constant $c$, MTF included. 

*Keywords:* Competitive analysis; On-line algorithms; List update problem; Move-to-front rule

## 1. Introduction

For the last years there has been a growing interest in the competitive analysis of on-line algorithms. Good examples are the list access and update problem, the $k$-server problem, the memory paging and dynamic allocation problems, etc. (see for instance [1, 3–6, 12, 14, 16]).

Competitive analysis, introduced by Sleator and Tarjan, measures the quality of an on-line algorithm by comparing it against an optimal off-line algorithm. Recall that an *on-line* algorithm serves a sequence of requests (for instance, to retry information from a given data structure) attending one each time, and without prior knowledge of

* Corresponding author.
*E-mail address*: conrado@lsi.upc.es (C. Martínez).

[1] This research was supported by the EC ESPRIT BRA Program 20244 (ALCOM-IT), CI-CYT TIC97-1475-CE, DGES PB95-0787 (KOALA), and CIRIT 1997SGR-00366 (SGR). The first author was also supported by ACI-CONICYT DOG 2320 (Catalonia-Chile Cooperation Program) and CYTED (RITOS Network). The second author was also supported by a CIRIT FI grant (Comissió Interdepartamental de Recerca i Innovació Tecnològica). Part of this work was done while the second author visited Princeton University.

0304-3975/00/$ - see front matter 
PII: S0304-3975(98)00264-3

the future requests of the sequence. On-line algorithms for min-cost problems try to do their best to guess possible future requests and spot regularities and patterns that appear in the sequence, in order to keep the total cost of attending the sequence to a minimum. In contrast, *off-line* algorithms are provided with full knowledge of the sequence of requests to be served, and can exploit this knowledge to serve it optimally.

An off-line algorithm is said to be *optimal* if it serves any sequence of requests with minimal total cost. An on-line algorithm is defined to be *c-competitive* if the total cost it incurs to serve any (sufficiently long) sequence of requests is, at most, $c$ times the cost of serving the same sequence with an optimal off-line algorithm. More precisely, an on-line algorithm $\mathscr{A}$ is $c$-competitive if and only if there exists a constant $d$ such that for any sequence of requests $\sigma$,

$$C_{\mathscr{A}}(\sigma) \leqslant c \cdot C_{OPT}(\sigma) + d,$$

where $C_{\mathscr{A}}(\sigma)$ denotes the total cost of algorithm $\mathscr{A}$ to attend $\sigma$, and $C_{OPT}(\sigma)$ denotes the total cost of an optimal off-line algorithm to attend $\sigma$. The least such constant $c$ is called the *competitiveness ratio* of the on-line algorithm.

Another possible approach to the study of on-line algorithms is the so-called distributional analysis (also known as average-case analysis). Its goal is to compute the expected cost to serve a sequence of requests not known in advance, under some distributional hypothesis. For instance, it is often assumed that the requests are independently generated, so that a request of type $k$ is generated with probability $p_k$, irrespective of the time the request is made, and irrespective of previous or future requests. Typically, the on-line algorithm is compared against an off-line algorithm that knows the probabilistic properties of the source that generates the requests, but not the actual sequence. Many authors have claimed that competitive analysis provides stronger and more valuable results than distributional analysis, since the former does not rely upon any assumption about the sequence of requests.

One of the earliest and most often cited results in competitive analysis of on-line algorithms is the study of the move-to-front rule (MTF) for the list access and update problem in linked lists [16]. An algorithm for the list access problem attends a sequence of requests by traversing an unsorted linked list until it finds the sought item. In order to minimize the total cost of serving the requests, the algorithm is allowed to reorganize the list, exchanging the order of its items.

The MTF rule moves the last accessed (the last inserted) item to the front of the list; the rationale behind this rule is that recently accessed items may be probably requested in the near future. Several previous studies had shown that MTF performs very well in practice and in theory [2, 7, 15]. In particular, Bentley and McGeoch [2] showed that MTF attends any sequence of requests with cost at most twice that of an off-line algorithm which, before serving the requests, sorted the list decreasingly w.r.t. to the number of future requests to each item, and made no reorganization afterwards.

The seminal paper of Sleator and Tarjan [16] introduced competitive analysis proving that MTF's cost is at most twice the optimal cost on any input sequence. They

also proved that no deterministic on-line algorithm can perform better than MTF, thus providing a theoretical validation of the use in practice of MTF.

Sleator and Tarjan's work opened a vast new area, namely, the competitive analysis of on-line algorithms. A great number of related papers considered variants of the original list access and update problem. Particularly successful is the use of randomization, giving 1.75-competitiveness (and even smaller competitiveness ratios) thus beating the lower bound of 2 for the competitiveness ratio of deterministic on-line algorithms [1, 8, 14].

However, we think that there are some limitations in those results that have gone unnoticed for the last years. To the best of our knowledge, nobody has provided sound arguments for (nor against) the original model of costs for the list access and update problem. A thorough justification for this model (we will call it the *basic cost model*, from now on) is missing in [16] and the subsequent literature. There have been several attempts to generalize some of the assumptions in the basic cost model, but all them maintain its fundamental structure and limitations. Moreover, it seems that it has been implicitly assumed that no off-line algorithm can beat MTF by more than a constant factor, *for any reasonable cost model* ("knowledge of future operations cannot significantly help to reduce the cost of current operations" [16, p. 205]).

But, as we shall show hereafter, it turns out that there is such a reasonable cost model for which there cannot exist any $c$-competitive on-line algorithm for the list access and update problem, for any constant $c$.

Next sections are organized as follows. In Section 2 we point out two important limitations of the basic cost model, and present an alternative cost model. In Section 3 we present a divide-and-conquer off-line algorithm that achieves $\Theta(\log n)$ amortized cost per access when serving any sequence of $n$ different requests over a list with $n$ items. In Section 4 we prove a trivial $\Theta(n)$ lower bound for the worst-case amortized cost per access of any on-line algorithm. This implies that, under our model of costs, no on-line algorithm can be $c$-competitive for any constant $c$, MTF included. We also prove a lower bound $\log_2 n + o(\log n)$ for the amortized cost of any (off-line) algorithm, and present another divide-and-conquer algorithm that achieves this bound. The paper ends with some final conclusions in Section 5.

## 2. The cost model

The basic cost model presented in [16] is as follows (we will only consider the static list model, i.e. when no updates are allowed). Accessing the $i$th item costs $i$. Immediately after an access to the $i$th item, we are allowed to move it at no cost to any position closer to the front of the list. This is called a *free exchange*. Any other update in the list must consist in exchanges of consecutive items. Each of these exchanges is called a *paid exchange*, and costs 1.

The first thing one could argue is that the cost of traversing a link can differ from the cost of an exchange. This observation has been already considered, and it is not

difficult to see that taking into account this fact does not yield significantly different conclusions.

On the other hand, it is clear that there cannot be free exchanges under a realistic model. However, while traversing the list in order to find the sought item, it is simple to keep a pointer at some position nearer to the beginning of the list, and update the list at small constant cost. Again, considering that free exchanges have in fact some constant cost (say 1) does not affect the main conclusions of [16] and most other papers on this subject. So this is not the point, either.

There are two reasons that, in our opinion, make the basic cost model not utterly realistic. First of all, in practice, there is no way of exchanging two consecutive items if they have not been reached before. Hence, we cannot consider that exchanging two consecutive items has cost 1, unless they appear in the list before the requested item. Otherwise, we should pay for the cost of accessing the items to be exchanged, plus one for the exchange itself.

Secondly, after reaching the currently requested item, any item before it in the list could be moved closer to the beginning of the list by means of a free (or constant time) exchange, for the same reason that we are allowed to use free exchanges with the requested item and move it closer to the front if we wish so. In other terms, the cost of moving any item before the requested one $j$ positions closer to the front of the list should not cost $j$ units, but a constant amount of time. This implies that, in our cost model, the cost of an access to the $k$th element plus the cost of the eventual reorganization of the items before that element is always *linear* in $k$ (with a constant that could be higher than that of the access to the $k$th element, but this fact cannot change the main results on competitiveness). This is consistent with the observation that, in a realistic setting, a complete rearrangement of all the items of a list before a certain element would require time proportional to the distance of the element to the front of the list (recall that we only measure the cost of traversing and updating pointers, not the cost of the computations to decide which pointers must be traversed or updated). In sharp contrast, the basic cost model charges *quadratic* cost for this complete rearrangement. In particular, assuming the basic cost model implies that sorting a linked list with $n$ items costs in general $\Theta(n^2)$, while sorting it using mergesort, for instance, costs only $\Theta(n \log n)$ – and we can even sort it with $\Theta(n)$ cost if we only measure the number of pointers traversed or updated, which is the usual quantity of interest in the list access problem.

The first of the two reasons above against the basic cost model does not affect the main results on competitiviness. Considering more realistic costs for paid exchanges, as explained above, actually means that the off-line algorithm would have to pay more than it was charged by the basic cost model, while the cost of any of the on-line algorithms presented so far (MTF included) would remain the same as they do not use paid exchanges.

But the second reason above turns out to change things dramatically. In the next section we will see how a simple off-line algorithm can profit by these generalized free (constant time) exchanges to defeat any on-line algorithm.

Our main objection to the basic cost model is thus that the seemingly omnipotent off-line algorithm has to pay a prohibitive cost if it wants to take advantage of its knowledge. In the basic cost model, it is not worth the effort to do anything but the same type of operations that MTF does, once a convenient ordering of the list has been achieved. Hence, the conclusion that MTF is competitive follows – basically, this is the result of Bentley and McGeoch – but we should be aware that MTF is competitive against a rather weakened off-line algorithm.

In other contexts, for instance, in the study of on-line paging algorithms, restrictions on the almighty off-line algorithm/adversary are achieved by means of the so-called access graphs (see for instance [3, 9]), and the strength and limitations of the results are fairly well understood. In the same context, Torng [17] proposed a cost model different from the traditional one, which led to significantly different results. On the other hand, some authors have criticized the definition of competitive ratio, and discussed ways to modify the notion of competitiveness to make it more realistic [10, 11, 13, 19].

## 3. An off-line algorithm

In this section we present an off-line algorithm that achieves $\Theta(n \log n)$ cost to serve *any* sequence of $n$ different requests to access a list with $n$ elements. Thus, we assume that the sequence of requests is, in fact, a permutation of the items in the list. This restriction is enough for our purposes, and there is no loss of generality. In particular, the arguments that we give here can be easily generalized to sequences of length $m$, consisting of $m/n$ blocks of requests, each one a permutation of the $n$ elements of the list.

Roughly speaking, our off-line algorithm (see Fig. 1) works as follows. To serve the first request, the algorithm visits all the nodes of the list, irrespective of the requested item. Although the algorithm pays $n$ at this step, it can save much of the future work by means of a simple divide-and-conquer strategy, which consists in splitting the original list into three sublists, one with the requested item, and two additional sublists with $(n-1)/2$ elements each. The first one of these sublists, `first`, contains the items that will be requested first, the other, `second`, the items that will be requested afterwards. The new list is built by joining `first`, `second` and the requested item in this order. This strategy ensures that the following $(n-1)/2$ requests will hit elements in the first half of the list. Therefore, these requests can be recursively served by the same algorithm, but now over a sublist half the size of the original one. This is the main idea of the algorithm.

When all the elements in the first sublist have been already accessed, the next requests will be for the items in the second sublist. The first of these requests is served by

1. traversing `previous`, the sublist with the $(n-1)/2$ previously accessed items (the old `first`, already served);

```
  /* sigma is a global variable with the sequence to be served; t is another
     global variable that indicates the next request to be attended. */
serve(list, sz, previous)
{
  /* Traverse and disconnect the first sz items in list, building three new lists by
     adding elements at their ends:
       • first: contains the (sz − 1)/2 items that, according to sigma,
       will be accessed first;
       • second: contains the (sz − 1)/2 items that will be accessed afterwards;
       • current: contains sigma[t], the currently requested item.
     Finally, return rest, a pointer to the (sz + 1)th item of the old list. */
classify(list, sz, &first, &second, &current, &rest);
  /* Service the request to the item pointed to by current. */
attend(current); t = t + 1;
  /*Concatenate the lists first, second, current, previous and rest, in this
     order, and return list, a pointer to the head of the so constructed linked list. */
join(first, second, current, previous, rest, &list);
if (sz == 1) return;
serve(list, (sz - 1)/2, NIL);
  /*Traverse the first (sz − 1)/2 items in list (just served). These items form
     a new linked list, called previous. After the call, list points to the
     beginning of the remaining elements. */
chop(&list, (sz − 1)/2, &previous);
serve(list, (sz − 1)/2, previous);
}
```

Fig. 1. The first off-line algorithm.

2. acting exactly as described for the general case but over the next $(n-1)/2$ items (that is, splitting this sublist into three sublists and joining them), and
3. joining the result of the step above with `previous`.

Again, the new list has at its front a sublist with the $(n-1)/2$ elements that are going to be immediately requested (except for the farthest of them, which is the one just served). Hence, we can use the same algorithm recursively to serve the second sublist and end up attending all the requests.

Let $L$ be the original list to be accessed. For the sake of simplicity, let us assume $n=2^m-1$ for some $m\geqslant 1$. We give a recursive version of the off-line algorithm `serve(list, sz, previous)`. The meaning of the parameters is as follows: `list` is a pointer to the beginning of the linked list; $\mathtt{sz}=2^k-1$ is the number of items to be attended by the current recursive call to `serve`, for some $1\leqslant k\leqslant m$; `previous` is a pointer to a list of already serviced items that have been just traversed and must be

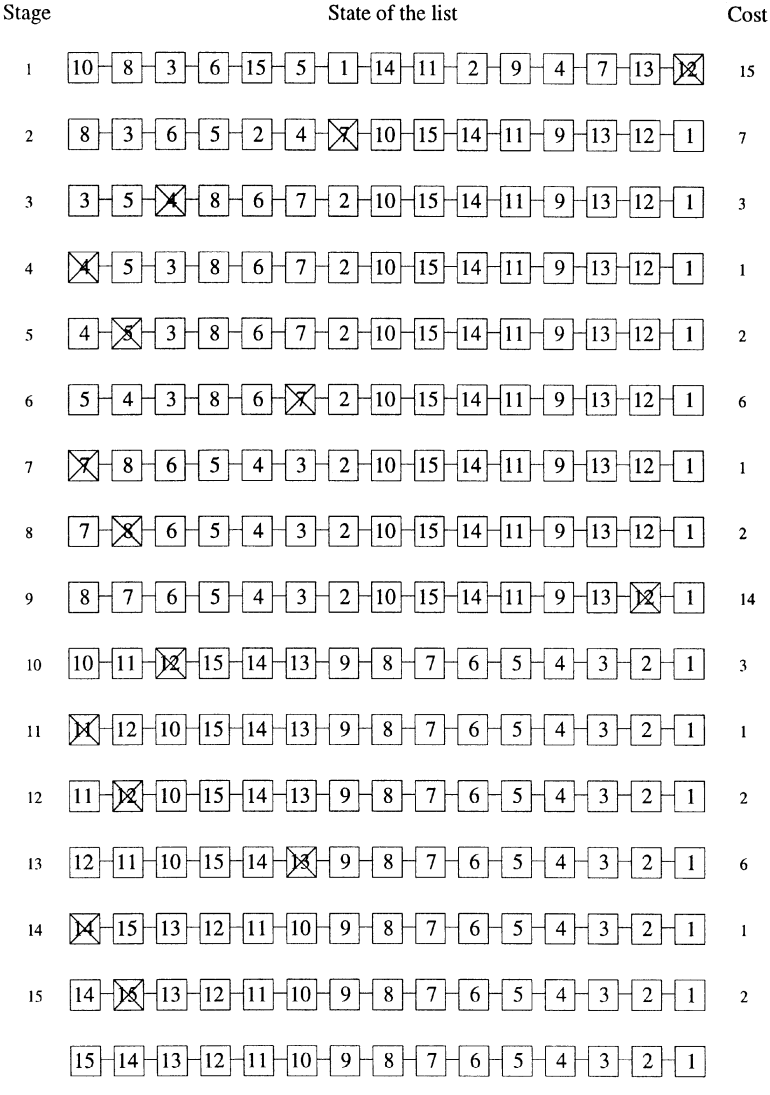


Fig. 2. An execution of the first off-line algorithm.

added after the first `sz` elements in `list` (the elements that are going to be served). The first call to `serve` is thus `serve(L, n, NIL)`.

We assume that the sequence of requests is stored in a global variable `sigma`, accessible to all the procedures below. Requests are served one at a time, and the global variable `t`, which is initialized to 1, indicates the next request to be attended.

A possible execution of this algorithm is given in Fig. 2. In the example, the sequence of requests is $\sigma = 1, 2, \ldots, n$ with $n = 15$. Each row reflects the order of the items in the linked list at each stage, the initial row being the initial state of the list and each

successive row showing one step of the process. The farthest element to the right reached during each stage is marked. The last column keeps track of the cost of each stage, in this case, the number of visited items.

Notice that every traversal of the list always begins at the first item of the linked list: the off-line algorithm never uses extra pointers to skip several items at a time or access the $i$th item paying less than $i$ units of cost. In other words, there is no trick in the use of recursion.

The analysis of the cost of the algorithm is quite simple. Let $T(k)$ be the number of visited nodes while serving a sequence with $2^k-1$ requests. Clearly, $T(1)=1$. Furthermore, for every $k\geqslant 2$ we have to pay $2^k-1$ for visiting all the nodes of the list, plus the cost to serve the first $2^{k-1}-1$ requests, plus $2^{k-1}-1$ for traversing `previous`, plus the cost to serve the last $2^{k-1}-1$ requests. This yields the recurrence

$$\begin{aligned} T(k) &= (2^k-1)+T(k-1)+(2^{k-1}-1)+T(k-1) \\ &= 3\cdot 2^{k-1}-2+2T(k-1). \end{aligned}$$

A simple proof by induction yields $T(k)=(3k-4)2^{k-1}+2$. We can use this result to compute the amortized cost to serve a request in the whole list as

$$\frac{T(m)}{n}=\frac{3m}{2}-2+\frac{3m}{2^{m+1}-2}=\frac{3}{2}\cdot\log_2 n+\mathcal{O}(1)=\Theta(\log n).$$

Finally and to be completely rigorous, we should mention that we have not shown which is the cost of deciding whether an item must be attached to `first` or `second`, while we are classifying the list. First, we have not to, since internal computations made by off-line algorithms are not usually taken into account; only the cost of serving requests and reorganizing the list has to be considered. But even if we took these costs into account, it is not difficult to see that the amortized cost per access of our off-line algorithm is still $\Theta(\log n)$, if we let the algorithm organize the information in `sigma` as a hash table or search tree, and we measure the cost of any access to the information stored in `sigma` as $\Theta(1)$ or $\mathcal{O}(\log n)$, as usual.

Concerning the space, observe that the off-line algorithm uses constant auxiliary space per recursive call; this amounts to $\Theta(\log n)$ auxiliary space to attend the whole sequence of requests. Notice that if we were to achieve $\mathcal{O}(\log n)$ cost per access by building a search tree with the items of the linked list, then we would need $\Theta(n)$ additional pointers. Thus we may argue that the off-line algorithm is not even using too much extra space.

## 4. Lower bounds

In this section we prove a linear lower bound for the cost of any on-line algorithm (either deterministic or randomized) dealing with a sequence of $n$ different requests to access a list with $n$ elements. This lower bound, together with the cost of the off-

line algorithm presented in Section 3, implies that there are no competitive on-line algorithms for the list access problem (under our cost model).

We also prove a lower bound $\log_2 n + o(\log n)$ for the cost of any off-line algorithm attending a sequence of requests that is a permutation of the $n$ elements of the list, and present another off-line algorithm that achieves this bound.

**Theorem 4.1.** *Let $\mathscr{A}$ be any on-line algorithm for the list access problem. For every list of size $n \geqslant 1$, there is always a sequence of $n$ different requests that forces $\mathscr{A}$ to spend $\Theta(n)$ (expected, if the algorithm is randomized) amortized cost per access to serve the sequence.*

**Proof.** If the on-line algorithm is deterministic, then it is trivial to enforce $\Theta(n)$ cost per access. It suffices that the adversary asks for the last item in the list which has not been requested yet. This simple strategy induces a cost to service the sequence $n + (n-1) + \cdots + 1 = n(n+1)/2 = \Theta(n^2)$; therefore, the worst-case amortized cost of $\Theta(n)$ per access follows.

For randomized algorithms, the lower bound can be easily proven using Yao's lemma [18]: we just need to show that the best deterministic on-line algorithm for the worst possible probability distribution over the sequence of requests incurs $\Theta(n)$ amortized expected cost per access.

Take the uniform probability distribution over the set of permutations of the $n$ items of the list. Then the average cost to serve the first request will be $(n+1)/2$, irrespective of what the on-line algorithm does and of the initial configuration of the list, as each element has identical probability of being the first to be requested. Thus the adversary can force even the best algorithm to pay expected cost $\Theta(n)$ to serve the first request in a list with $n$ items.

After that, the best thing to happen from the on-line algorithm's point of view is that it moves the accessed item to the end of the list, since this item will not be requested any more, and the other $n-1$ items will be requested with identical probability in the second round. So we are in the same situation as before, except that, from the point of view of the adversary, the list has now $n-1$ elements. This argument shows that there is a distribution over the sequences of requests that induces a total expected cost $(n+1)/2 + n/2 + \cdots + 1 = n(n+3)/4$ even for the best algorithm, and hence, that $\Theta(n)$ per access is a lower bound for the expected amortized cost of any randomized on-line algorithm. □

**Theorem 4.2.** *Let $L_n$ be the minimum number of visited items while serving* any *permutation of $n$ elements with the best off-line algorithm (whatever it is). Then $L_n \geqslant (n+1)\log_2(n+1) - n$.*

**Proof.** We can prove the theorem by induction. When $n = 1$, then we have $L_1 = 1 = 2\log_2 2 - 1$.

Let us now assume that the theorem is true for every size $< n$, and that we are given a permutation with $n$ items. Irrespective of the permutation to be served, it is clear that

| $n$ | Optimal pattern | $L_n$ | Lower bound |
|---|---|---|---|
| *1* | 1 | *1* | *1* |
| *2* | 1 2 | *3* | *2.75⋯* |
| *3* | 1 3 1 | *5* | *5* |
| *4* | 1 2 4 1 | *8* | *7.60⋯* |
| *5* | 1 2 5 1 2 | *11* | *10.50⋯* |
| *6* | 1 3 1 6 1 2 | *14* | *13.65⋯* |
| *7* | 1 3 1 7 1 3 1 | *17* | *17* |

Fig. 3. Some optimal patterns.

the off-line algorithm has to reach the last position of the list at least once. Let $k$ be the number of requests served by the algorithm before the first visit to the final position of the list, $0 \leqslant k < n$. Then, the minimum cost to attend $k$ requests —as if there were no elements in the list but these $k$— plus $n$ (the cost to serve the $(k+1)$-th request), plus the minimum cost to attend the last $n-1-k$ requests —as if these elements were alone in the list— is a lower bound for $L_n$. That is, $L_n \geqslant L_k + n + L_{n-1-k}$. Therefore, we have

$$\begin{aligned} L_n &\geqslant \min\{0 \leqslant i < n\colon L_i + n + L_{n-1-i}\} \\ &\overset{\text{i.h.}}{\geqslant} n + \min\{0 \leqslant i < n\colon (i+1)\log_2(i+1) - i + (n-i)\log_2(n-i) - (n-1-i)\} \\ &= 1 + \min\{0 \leqslant i < n\colon (i+1)\log_2(i+1) + (n-i)\log_2(n-i)\}. \end{aligned}$$

Taking into account that the function over the reals $f(x) = (x+1)\log_2(x+1) - (n-x)\log_2(n-x)$ achieves its minimum at $(n-1)/2$, we get

$$L_n \geqslant 1 + f\left(\frac{n-1}{2}\right) = (n+1)\log_2(n+1) - n.$$

The theorem follows. □

Fig. 3 shows several optimal patterns, together with the cost of the pattern and the value corresponding to the lower bound $(n+1)\log_2(n+1) - n$. For instance, assume $n=4$ and that the sequence of requests is *a, b, c, d*. If the initial configuration of the list is (*a b d c*), then we can attend the sequence of requests visiting the first, second and fourth positions, placing *d* at the front, and finally visiting the first position, that is, following the pattern 1 2 4 1. Moreover, it is easy to see that there is not any other pattern that provides smaller cost. Note that, in general, there are several optimal patterns for one fixed $n$.

Finally, we provide an off-line algorithm that, under our cost model, attends a sequence of $n$ different requests with a total cost of $n\log_2 n + o(n\log n)$ and, therefore, is nearly optimal (see Fig. 4 for an execution of the algorithm). Recall that the algorithm given in the previous section attends such a sequence with total cost $\frac{3}{2}n\log_2 n + o(n\log n)$.

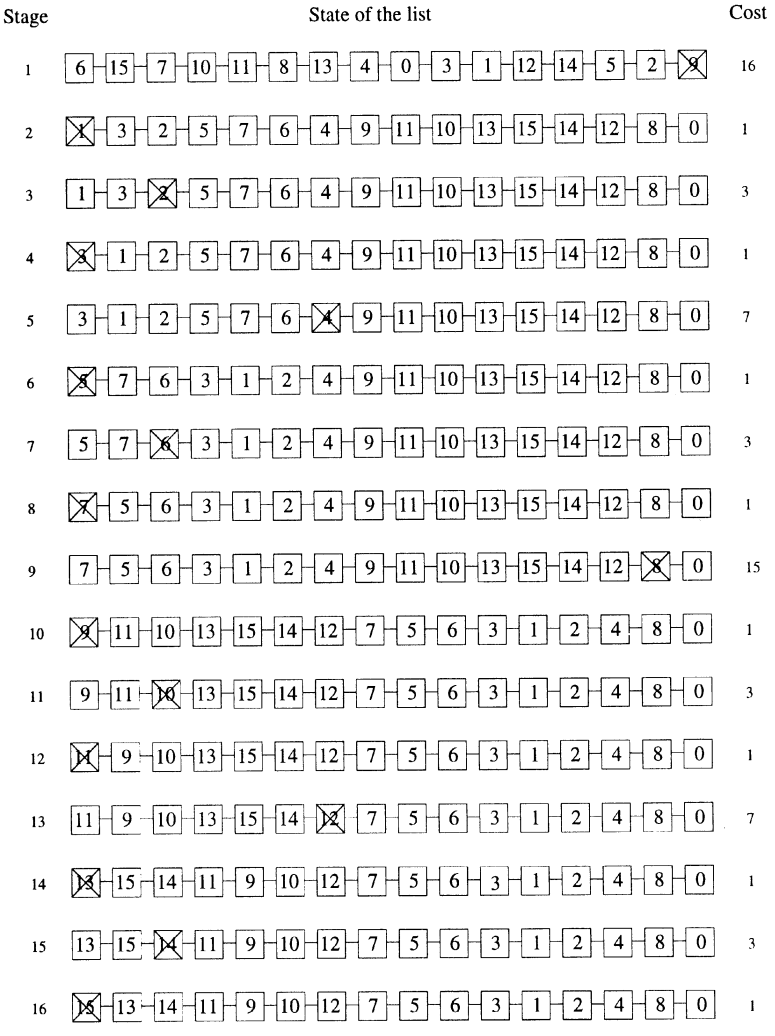


Fig. 4. An execution of the second off-line algorithm.

For the sake of simplicity, let $n = 2^m$, where $m \geqslant 0$, and assume that the sequence of requests is $\sigma = 0, 1, \ldots, n-1$. Consider the following algorithm: To attend the first request (to 0), skip the whole list, and reorganize it completely, building a permutation that minimizes the time to serve the rest of $2^m - 1$ requests. The construction of this permutation follows the divide-and-conquer paradigm, and consists in joining the optimal permutation for the first $2^{m-1} - 1$ elements, the optimal permutation for the last $2^{m-1} - 1$ elements, and finally the $2^{m-1}$th element.

After this optimal configuration is achieved, the rest of requests are trivially attended with minimal cost. The total time of the off-line algorithm is thus $n + n\log_2 n - (n-1) = n\log_2 n + 1$.

In a practical setting, one could argue that the first step —the total reorganization of the list – could cost more than $\Theta(n)$, if we consider a completely realistic cost model (and thus we take into account even the internal computations to decide the new location of every item). In contrast, notice that the off-line algorithm given in Section 3 never performs such a drastic reorganization.

## 5. Conclusions

We have considered the list access problem and proved that, under a realistic cost model, no on-line algorithm can be $c$-competitive for any $c$. In particular, we have shown that the traditional result about the 2-competitiveness of MTF follows from the high costs associated in the basic cost model to the type of operations that would enable an off-line algorithm to take advantage of its full knowledge on the sequence of requests.

One simple way to state our objection to the basic cost model is that $\Theta(n^2)$ inversions can be removed in practice with cost $\Theta(n)$, once we allow for typical operations over linked lists and their costs are accounted as usual, but this costs $\Theta(n^2)$ in the basic cost model.

We think that our contribution is twofold. On the one hand, we can conclude that the knowledge on the future sequence of requests indeed improves the average time per access in the context of the list access and update problem.

On the other hand, although competitive analysis is a worthwhile technique for the study of on-line problems, we should carefully scrutinize which assumptions the competitiviness results are based upon. In particular, we claim that the basic cost model is interesting and helps explaining the good properties of MTF, but we must be aware of the limitations of the model and not exploit its weak points for the design of new list algorithms. Recall that MTF was devised (and already shown to perform well in practice) before the basic cost model was ever proposed, so our objections do not apply to the MTF rule itself.

A new open problem raised by our investigation is whether there exists some *on-line* algorithm that uses non-local exchanges and does much better than MTF. That is, if there exist some algorithm $\mathscr{A}$ and constant $c$ such that, for every sequence of requests $\sigma$ large enough, $C_{\mathscr{A}}(\sigma) \leqslant c \cdot C_{\mathrm{MTF}}(\sigma)$, but for every constant $c'$ there always exists some $\sigma$ large enough such that $C_{\mathrm{MTF}}(\sigma) \not\leqslant c' \cdot C_{\mathscr{A}}(\sigma)$. Our conjecture is that such on-line algorithm does not exist. Basically, we conjecture that no on-line algorithm could take much advantage of moving non-requested items far away from their positions towards the front of the list. Essentially, there would exist sequences such that MTF is much better than the hypothetical on-line algorithm $\mathscr{A}$ and viceversa, making them incomparable from the point of view of competitiveness.

Another (more important) open question is whether there exist alternative ways to define competitiveness such that MTF and other good on-line algorithms for the list update problem would be competitive, even for the cost model presented in this

work. Notice that the good performance of MTF comes from the regularities and patterns that we usually find in practical situations. Hence, one possibility could be to add restrictions over the sequences of requests, instead of weakening the cost model. It seems a rather difficult problem.

## Acknowledgement

We would like to thank Josep Díaz, Micha Hofri, Robert Tarjan and the anonymous referees for their useful comments and suggestions on the preliminary versions of this work.

## References

[1] S. Ben-David, A. Borodin, R. Karp, G. Tardos, A. Wigderson, On the power of randomization in on-line algorithms, Algorithmica 11 (1) (1994) 2–14.
[2] J.L. Bentley, C.C. McGeoch, Amortized analyses of self-organizing sequential search heuristics, Comm. ACM 28 (4) (1985) 404–411.
[3] A. Borodin, S. Irani, P. Raghavan, B. Schieber, Competitive paging with locality of reference, J. Comput. System Sci. 50 (2) (1995) 244–258.
[4] A. Borodin, N. Linial, M. Saks, An optimal on-line algorithm for metrical task systems, J. ACM 39 (4) (1992) 743–763.
[5] A. Fiat, R.M. Karp, M. Luby, L.A. McGeoch, D.D. Sleator, N.E. Young. Competitive paging algorithms, J. Algorithms 12 (1991) 685–699.
[6] A. Fiat, Y. Rabani, Y. Ravid, Competitive $k$-server algorithms, J. Comput. System Sci. 48 (3) (1994) 410–428.
[7] G. Gonnet, J.I. Munro, H. Suwanda, Exegesis of self-organizing linear search, SIAM J. Comput. 10 (3) (1981) 613–637.
[8] S. Irani, Two results on the list update problem, Inform. Process. Lett. 38 (1991) 301–306.
[9] S. Irani, A.R. Karlin, S. Phillips, Strongly competitive algorithms for paging with locality of reference, SIAM J. Comput. 25 (3) (1996) 477–497.
[10] C. Kenyon, Best-fit bin-packing with random order, in: ACM-SIAM Symp. on Discrete Algorithms (SODA), 1996, pp. 359–364.
[11] E. Koutsoupias, C.H. Papadimitriou, Beyond competitive analysis, in: Proc. 35th Annu. Symp. on Foundations of Computer Science, 1994, pp. 394–400.
[12] M.S. Manasse, L.A. McGeoch, D.D. Sleator, Competitive algorithms for server problems, J. Algorithms 11 (1990) 208–230.
[13] P. Raghavan, M. Snir, Memory versus randomization in on-line algorithms, IBM J. Res. Dev. 38 (6) (1994) 683–707.
[14] N. Reingold, J. Westbrook, D.D. Sleator, Randomized competitive algorithms for the list update problem, Algorithmica 11 (1) (1994) 15–32.
[15] R.L. Rivest, On self-organizing sequential search heuristics, Comm. ACM 19 (2) (1976) 63–67.
[16] D.D. Sleator, R.E. Tarjan, Amortized efficiency of list update and paging rules, Comm. ACM 28 (2) (1985) 202–208.
[17] E Torng, A unified analysis of paging and caching, in: Proc. 36th Annu. Symp. on Foundations of Computer Science, 1995, pp. 194–203.
[18] A. C. Yao, Probabilistic computations: towards a unified measure of complexity, in: ACM Symp. on Theory of Computing (STOC), 1980, pp. 222–227.
[19] N.E. Young, Cross-input amoritzation captures the diffuse adversary, Technical Report PCS-TR96-302, Dartmouth College, 1996.