Intro to Analysis of Algorithms
Online
Chapter 5

Michael Soltys

CSU Channel Islands

[**Git** Date:2018-11-20 Hash:f93cc40 Ed:3rd]

List Accessing

If a file is in position i, we incur a search cost of i in locating it.

If the file is not in the cabinet, the cost is l, which is the total number of files.

Let $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_n$ be a finite sequence of n requests. To service request σ_i, a list accessing algorithm ALG must search for the item labeled σ_i by traversing the list from the beginning, until it finds it.

The cost of retrieving this item is the index of its position on the list. Thus, if item σ_i is in position j, the cost of retrieving it is j. Furthermore, the algorithm may reorganize the list at any time.

The work associated with a reorganization is the minimum number of transpositions of consecutive items needed to carry it out.

Each transposition has a cost of 1, however, immediately after accessing an item, we allow it to be moved free of charge to any location closer to the front of this list.

These are *free* transpositions, while all other transpositions are *paid*.

Let $\mathrm{ALG}(\sigma)$ be the sum of the costs of servicing all the items on the list σ, i.e., the sum of the costs of all the searches plus the sum of the costs of all paid transpositions.

Let OPT be an optimal (offline) algorithm for the static list accessing model.

Suppose that OPT and MTF both start with the same list configuration. Then, for any sequence of requests σ, where $|\sigma| = n$, we have that

$$\mathrm{MTF}(\sigma) \leq 2 \cdot \mathrm{OPT}_S(\sigma) + \mathrm{OPT}_P(\sigma) - \mathrm{OPT}_F(\sigma) - n,$$

where $\mathrm{OPT}_S(\sigma), \mathrm{OPT}_P(\sigma), \mathrm{OPT}_F(\sigma)$ are the total cost of searches, the total number of paid transpositions and the total number of free transpositions, of OPT on σ, respectively.

Imagine that both MTF and OPT process the requests in σ, while each algorithm works on its own list, starting from the same initial configuration.

You may think of MTF and OPT as working in parallel, starting from the same list, and neither starts to process σ_i until the other is ready to do so.

Let

$$a_i = t_i + (\Phi_i - \Phi_{i-1})$$

where t_i is the actual cost that MTF incurs for processing this request (so t_i is in effect the position of item σ_i on the list of MTF *after* the first $i-1$ requests have been serviced).

Φ_i is a *potential function*, and here it is defined as the number of *inversions* in MTF's list with respect to OPT's list.

An inversion is defined to be an ordered pair of items x_j and x_k, where x_j precedes x_k in MTF's list, but x_k precedes x_j in OPT's list.

Note that Φ_0 depends only on the initial configurations of MTF and OPT, and since we assume that the lists are initially identical, $\Phi_0 = 0$.

Finally, the value a_i is called the *amortized cost*, and its intended meaning is the cost of accessing σ_i, i.e., t_i, plus a measure of the increase of the "distance" between MTF's list and OPT's list after processing σ_i, i.e., $\Phi_i - \Phi_{i-1}$.

It is obvious that the cost incurred by MTF in servicing σ, denoted $\mathrm{MTF}(\sigma)$, is $\sum_{i=1}^{n} t_i$.

But instead of computing $\sum_{i=1}^{n} t_i$, which is difficult, we compute $\sum_{i=1}^{n} a_i$ which is much easier.

The relationship between the two summations is,

$$\mathrm{MTF}(\sigma) = \sum_{i=1}^{n} t_i = \Phi_0 - \Phi_n + \sum_{i=1}^{n} a_i,$$

and since we agreed that $\Phi_0 = 0$, and Φ_i is always positive, we have that,

$$\mathrm{MTF}(\sigma) \leq \sum_{i=1}^{n} a_i.$$

So now it remains to compute an upper bound for a_i.

Assume that the i-th request, σ_i, is in position j of OPT, and in position k of MTF (i.e., this is the position of this item *after* the first $(i-1)$ requests have been completed). Let x denote this item.

We are going to show that

$$a_i \leq (2s_i - 1) + p_i - f_i,$$

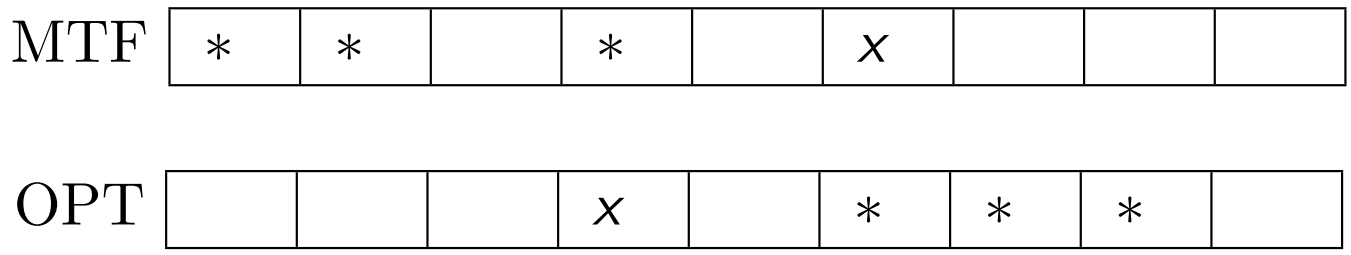
where s_i is the search cost incurred by OPT for accessing request σ_i, and p_i and f_i are the paid and free transpositions, respectively, incurred by OPT when servicing σ_i.

This shows that

$$\begin{aligned}\sum_{i=1}^{n} a_i &\leq \sum_{i=1}^{n}((2s_i - 1) + p_i - f_i) \\ &= 2(\sum_{i=1}^{n} s_i) + (\sum_{i=1}^{n} p_i) - (\sum_{i=1}^{n} f_i) - n \\ &= 2\mathrm{OPT}_S(\sigma) + \mathrm{OPT}_P(\sigma) - \mathrm{OPT}_F(\sigma) - n,\end{aligned}$$

Two prove our statement in two steps: in the first step MTF makes its move, i.e., moves x from the k-th slot to the beginning of its list, and we measure the change in the potential function *with respect to* the configuration of the list of OPT *before* OPT makes its own moves to deal with the request for x.

In the second step, OPT makes its move and now we measure the change in the potential function *with respect to* the configuration of the list of MTF *after* MTF has completed its handling of the request (i.e., with x at the beginning of the list of MTF).



x is in position k in MTF, and in position j in OPT. Note that in the figure it appears that $j < k$, but we make no such assumption in the analysis. Let $*$ denote items located before x in MTF but after x in OPT, i.e., the $*$ indicate inversions with respect to x. There may be other inversions involving x, namely items which are after x in MTF but before x in OPT, but we are not concerned with them.

suppose that there are v such $*$, i.e., v inversions of the type represented in the figure. Then, there are at least $(k-1-v)$ items that precede x in both list.

But this implies that $(k-1-v) \leq (j-1)$, since x is in the j-th position in OPT. Thus, $(k-v) \leq j$. So what happens when MTF moves x to the front of the list? In terms of inversions two things happen: (i) $(k-1-v)$ new inversions are created, with respect to OPT's list, before OPT itself deals with the request for x. Also, (ii) v inversions are eliminated, again with respect to OPT's list, before OPT itself deals with the request for x.

Therefore, the contribution to the amortized cost is:

$$k + ((k - 1 - v) - v) = 2(k - v) - 1 \overset{(1)}{\leq} 2j - 1 \overset{(2)}{=} 2s - 1$$

where (1) follows from $(k - v) \leq j$ shown above, and (2) follows from the fact that the search cost incurred by OPT when looking for x is exactly j.

In the second step of the analysis, MTF has made its move and OPT, after retrieving x, rearranges its list. Each paid transposition contributes 1 to the amortized cost and each free transposition contributes -1 to the amortized cost.

In the *dynamic list accessing model* we also have *insertions*, where the cost of an insertion is $l + 1$—here l is the length of the list—, and *deletions*, where the cost of a deletion is the same as the cost of an access, i.e., the position of the item on the list.

Our result still holds in the dynamic case.

The *infimum* of a subset $S \subseteq \mathbb{R}$ is the largest element r, not necessarily in S, such that for all all $s \in S$, $r \leq s$.

We say that an online algorithm is *c-competitive* if there is a constant α such that for all finite input sequences $\mathrm{ALG}(\sigma) \leq c \cdot \mathrm{OPT}(\sigma) + \alpha$.

The infimum over the set of all values c such that ALG is c-competitive is called the *competitive ratio* of ALG and is denoted $\mathcal{R}(\mathrm{ALG})$.

Observe that $\mathrm{OPT}(\sigma) \leq n \cdot l$, where l is the length of the list and n is $|\sigma|$.

MTF is a 2-competitive algorithm, and that $\mathcal{R}(\mathrm{MTF}) \leq 2 - \frac{1}{l}$.

Competitive analysis: the payoff of an online algorithm is measured by comparing its performance to that of an *optimal offline algorithm*

Competitive analysis thus falls within the framework of *worst case* complexity.

Paging

Consider a two-level *virtual memory system*: each level, slow and fast, can store a number of fixed-size memory units called *pages*. The slow memory stores N pages, and the fast memory stores k pages, where $k < N$. The k is usually much smaller than N.

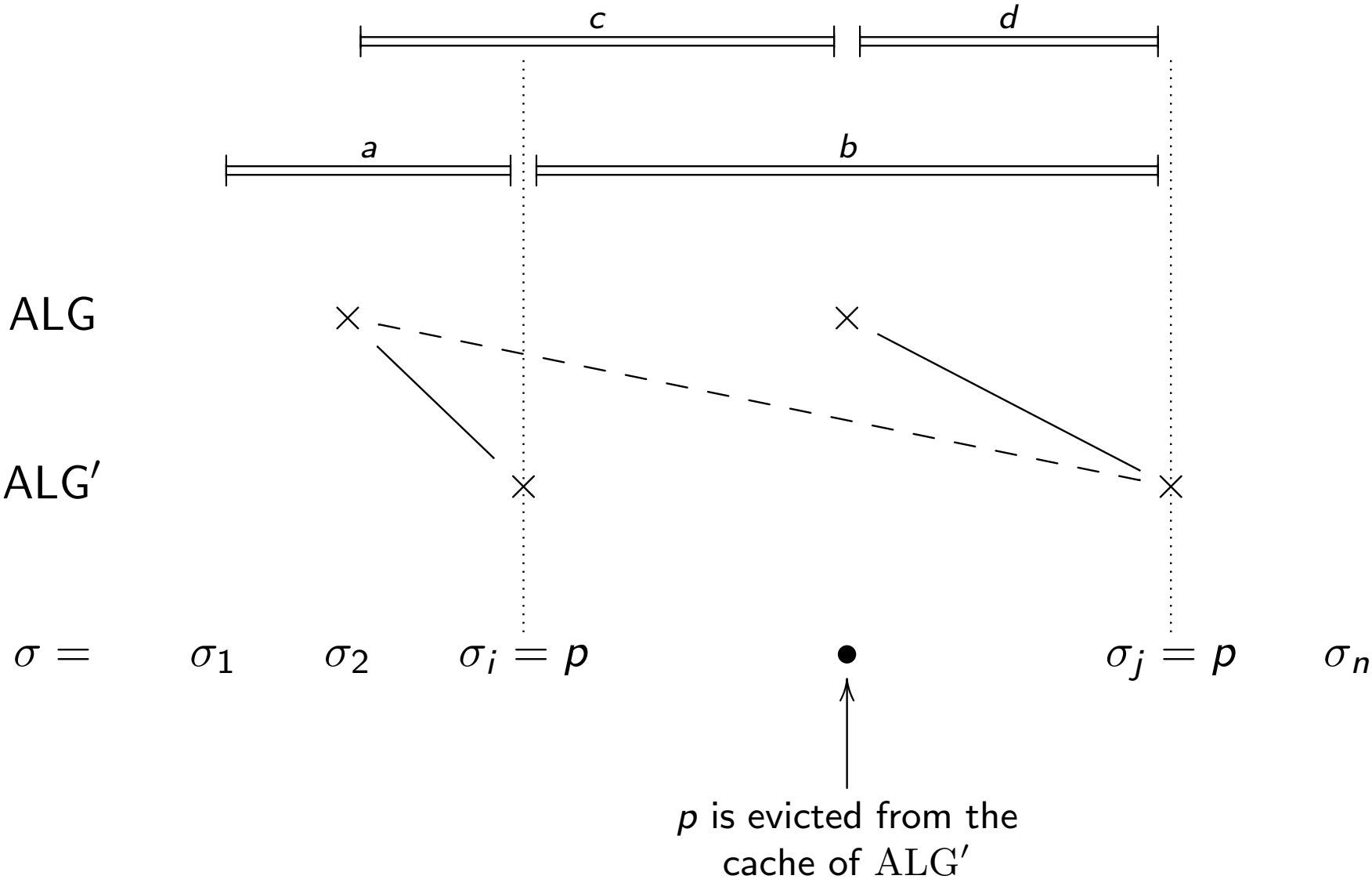
Given a request for page p_i, the system must make page p_i available in the fast memory. If p_i is already in the fast memory, called a *hit*, the system need not do anything. Otherwise, on a *miss*, the system incurs a *page fault*, and must copy the page p_i from the slow memory to the fast memory. In doing so, the system is faced with the following problem: which page to evict from the fast memory to make space for p_i. In order to *minimize* the number of page faults, the choice of which page to evict must be made wisely.

LRU	*Least Recently Used*
CLOCK	*Clock Replacement*
FIFO	*First-In/First-Out*
LIFO	*Last-In/First-Out*
LFU	*Least Frequently Used*
LFD	*Longest Forward Distance*

Demand paging algorithms never evict a page from the cache unless there is a page fault, that is, they never evict preemptively. All the paging disciplines in the table on the previous slide are demand paging.

We consider the *page fault model*, where we charge 1 for bringing a page into the fast memory, and we charge nothing for accessing a page which is already there.

Theorem 5.11: Any page replacement algorithm, online or offline, can be modified to be demand paging without increasing the overall cost on any request sequence.


c
d
a
b
ALG
ALG′
$\sigma =$
σ_1
σ_2
$\sigma_i = p$
$\sigma_j = p$
σ_n
p is evicted from the
cache of ALG'

Suppose that i, j is the smallest pair such that there exists a page p with the property that $\sigma_i = \sigma_j = p$. ALG' incurs a page fault at σ_i and σ_j, and the two corresponding page moves of ALG' are both matched with the same page move of p by ALG somewhere in the stretch a.

We show that this is not possible: if ALG' incurs a page fault at $\sigma_i = \sigma_j = p$ it means that somewhere in b the page p is evicted—this point is denoted with '•'.

If ALG did not evict p in the stretch c, then ALG also evicts page p at '•' and so it must then bring it back to the cache in stretch d—we would match the $\times$ at σ_j with that move.

If ALG did evict p in the stretch c, then again it would have to bring it back in before σ_j. In any case, there is a later move of p that would be matched with the page fault of ALG' at σ_j.

FIFO

When a page must be replaced, the *oldest* page is chosen. It is not necessary to record the time when a page was brought in; all we need to do is create a FIFO (First-In/First-Out) queue to hold all pages in memory.

The FIFO algorithm is easy to understand and program, but its performance is not good in general.

FIFO also suffers from the so called *Belady's anomaly*. Suppose that we have the following sequence of page requests: $1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5$. Then, we have more page faults when $k = 4$ than when $k = 3$. That is, FIFO has more page faults with a bigger cache!

LRU

If we use the recent past as an approximation of the near future, then we will replace the page that *has not been used for the longest period of time*. This approach is the *Least Recently Used (LRU)* algorithm.

LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time. The LRU algorithm is considered to be good, and is often implemented—the major problem is *how* to implement it; two typical solutions are counters and stacks.