6.897: Advanced Data Structures Spring 2003

Lecture 6 — March 3, 2003

Prof. Erik Demaine *Scribes: Patrick Lam & Ilya Shlyakhter*

1 Overview

In the previous lecture we discussed self-organizing list data structures, static optimality, and the move-to-front (MTF) and frequency-count (FC) algorithms for maintaining a list order which enables relatively fast searches. The lecture culminated with the claims:

$$\begin{aligned} \text{cost(MTF)} &\leq 2\,\text{cost(static OPT)} \qquad \text{(with proof)} \\ \text{cost(FC)} &\leq 2\,\text{cost(static OPT)} \end{aligned}$$

where cost measures the number of comparisons done by these algorithms.

In the algorithms considered so far, we have only one mutable pointer, which may point to any list element. This (vaguely) models caches and the least-recently-used algorithm for cache upkeep; however, cache hardware is generally much more sophisticated. More importantly, we are discussing self-organizing lists as a prelude to our forthcoming treatment of self-organizing tree structures.

Notation from previous lecture which is carried over to this lecture is as follows:

- m: number of requests in a sequence
- n: number of distinct elements (i.e., list length)

In this lecture we define the notion of dynamic optimality and show the dynamic optimality of move-to-front in the Sleator & Tarjan cost model. We continue by introducing the Munro model, a more permissive model in which some request sequences can run significantly faster than in the Sleator & Tarjan model.

2 Dynamic Optimality

Until now, we have been comparing to "static OPT", which is defined to be the optimal list order subject to the constraint that the list order cannot change in the middle of the request sequence. (In the stochastic model, this constraint didn't matter, because we assumed that the search requests were independent: since we weren't able to predict what was coming next, no algorithm could preemptively optimize the list order to take advantage of information about future requests.) In some sense, comparing the MTF and FC algorithms to static optimality is "cheating", because we are using dynamic information to approximate a particular static ordering.

Henceforth, we drop the independence assumption; we will talk about request sequences. If an algorithm has the entire future request sequence available to it, then the algorithm will be able to outpace a statically optimum algorithm. In our model, we allow the optimum algorithm to be *omniscient*. (The terminology suggests that such a model would be unreasonable, but we talk about these algorithms all the time; usually we call them *offline* algorithms.) We expect that OPT would re-order its list dynamically to use its information about the future request sequence. If we're lucky, we'll still be able to design non-omniscient (online) algorithms that are nearly as good as the optimal omniscient (offline) algorithm.

2.1 Survey of Extant Results

We next discuss several results about dynamic optimality.

2.1.1 Preliminaries

First, we need to define our terms; we shall evaluate our algorithms using the following definition:

Definition 1. *(Competitiveness) Algorithm A is* c-competitive *if there is a number b such that, for all request sequences σ,*

$$\mathrm{cost}_A(\sigma) \leq c \cdot \mathrm{cost}_{OPT}(\sigma) + b,$$

where OPT is an omniscient optimum algorithm.

That is, for all inputs σ, our online algorithm A runs at most c times slower than the best possible algorithm. Note that algorithm A is handicapped: it only learns the request sequence as the searches happen, while OPT is an omniscient algorithm; it has access to the entire request sequence ahead of time.

Cost model. We also need an explicit definition of "cost" to know what the optimum algorithm gets charged for. Here's the one used by Sleator and Tarjan in [ST85]:

- pay i to find element in position i;
- free to move found element anywhere closer to front (*free-swap*); and
- pay 1 per swap of two adjacent elements – no algorithms we've seen so far perform any such swaps (*paid-swap*).

2.1.2 Dynamic Optimality Results

Given a cost model and a definition of c-competitive, we can discuss the goodness of the algorithms we've seen so far. [ST85] showed the following:

- The Move-to-Front algorithm is 2-competitive.
- Frequency-Count is not $O(1)$-competitive.
- Transpose is not $O(1)$-competitive.

We will present the competitiveness proof for Move-To-Front shortly. Intuitively, Transpose is not $O(1)$-competitive because the request sequence `C D A B A B A B ...` will never bring either `A` or `B` to the front, giving an arbitrarily bad cost. Similarly, Frequency-Count can be foiled by a sequence of n `A`s, followed by $n-1$ `B`s, followed by $n-2$ `C`s, etc.

2.2 Beating Move-to-Front

Can we find an algorithm that is better than 2-competitive? The following unpublished result by Karp and Raghavan answers this question in the negative:

Theorem 2. *No deterministic algorithm is c-competitive, where $c < 2$.*

We can do better than 2-competitive, though, if our algorithm is not deterministic.

Bit Algorithm. Consider this randomized algorithm:

- Initially choose a random bit for each list element.
- Upon access to an element, flip its bit. If 1, MTF.

Theorem 3. *With an oblivious adversary model, in expectation of ratios, the Bit algorithm is 1.75-competitive [RWS94].*

OPEN: Is Bit better than 1.75-competitive? What about the variance of its expectation? In particular, does 1.75-competitiveness hold with high probability; *i.e.* for any α, is the probability is at least $1 - O(1/n^\alpha)$? Rob Seater asked: Can we do better by choosing the random bits carefully?

Related work: The best randomized algorithm so far is 1.6-competitive [Alb98]; it uses, in game-theoretic terms, a mixed strategy. No algorithm is better than 1.5-competitive [Tei93]. It is NP-hard to compute an optimum offline strategy [Amb00].

ALSO OPEN: Close the gap; there seems to be a lot of room to maneuver between 1.5- and 1.6-competitiveness. Other open problems can be found in Chapter 2 of [BEY98].

2.3 Proof of MTF Competitiveness Bound

We will now prove MTF's 2-competitiveness:

$$\forall \sigma.\ \text{cost}_{\text{MTF}}(\sigma) \le 2 \cdot \text{comp-cost}_{\text{OPT}}(\sigma) + \text{paid-swaps}_{\text{OPT}}(\sigma) - \text{free-swaps}_{\text{OPT}}(\sigma) - m.$$

In this inequality, $\text{free-swaps}_{\text{OPT}}(\sigma)$ refers to the number of "free swaps" taken by the dynamic optimal algorithm on σ (resp. "paid swaps"). The comp-cost term handles the amount of time spent searching for elements by the dynamic optimum algorithm. Note that this inequality is enough to show that MTF is competitive:

$$
\begin{aligned}
2 \cdot \text{comp-cost}_{\text{OPT}}(\sigma) + \text{paid-swaps}_{\text{OPT}}(\sigma) - \text{free-swaps}_{\text{OPT}}(\sigma) - m & \\
&\leq 2(\text{comp-cost}_{\text{OPT}}(\sigma) + \text{paid-swaps}_{\text{OPT}}(\sigma)) \\
&= 2 \cdot \text{cost}_{\text{OPT}}(\sigma)
\end{aligned}
$$

More precisely, MTF is amortized 2-competitive; our proof will show this using potential functions.

Defining our potential function. We will run MTF and OPT in parallel on the same request sequence, and the potential function Φ will represent our bank: how many operations we must pay for in the future.

Definition 4. *Our potential function is*

$$
\begin{aligned}
\Phi &= \#\ \textit{of inversions between MTF order and OPT order} \\
&= \#\ \textit{of pairs of elements } (i,j) \textit{ where } i \textit{ precedes } j \textit{ in MTF order but not OPT order.}
\end{aligned}
$$

Free swaps. Right after a search for element i, a free swap is a swap which moves i closer to the front. MTF always moves the just-searched-for element i to the front of the list; this creates inversions if OPT does not also move i to the front of the list. We will account for the added inversions later, when we discuss the effect of searching for an element. For now, we simply note that, if OPT makes a free swap, Φ will decrease by 1: we've removed one of the just-added inversions.

Consider the following brief example; we'll assume that we have a singleton request sequence, $\langle \texttt{C} \rangle$.

	List	Φ
MTF:	`A B C`	0
OPT:	`A B C`	0
request `C`		
MTF moves `C` to front:		
MTF:	`C A B`	2
OPT:	`A B C`	
first free-swap by OPT:		
MTF:	`C A B`	1
OPT:	`A C B`	

Paid swaps. MTF does no paid swaps, so if OPT ever does a paid swap, we charge an increase of 1 to Φ: we are potentially introducing an inversion into OPT which wasn't in MTF.

These arguments account for the paid-swap and free-swap terms in Definition 4. We must still deal with the comp-cost term.

2.3.1 Cost of searching for i

How much does a search operation cost?

- Suppose that i is in position K_{OPT} in the OPT list, and position K_{MTF} in the MTF list.
- Suppose that, out of $K_{\mathrm{MTF}}-1$ elements in front of i in the MTF list, there are f elements in front of i in OPT, and b elements behind i in OPT. Note that $f+b=K_{\mathrm{MTF}}-1$.

  Figure 1 illustrates the situation; observe that b accounts for the i-generated inversions, while the f nodes don't contribute any i-generated inversions between MTF and OPT.

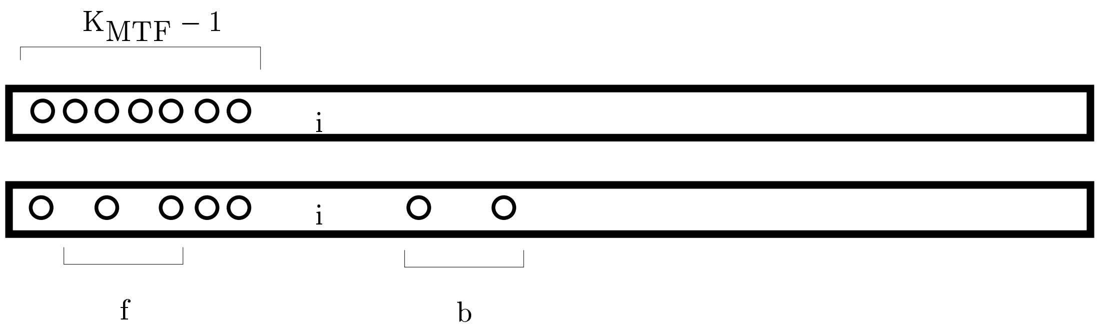

Figure 1: MTF versus OPT

- After MTF, all $K_{\mathrm{MTF}}-1$ of these elements go behind i in the MTF list. Hence:

$$\Phi \text{ increases by } f-b$$

  This is because each b was an inversion, but is no longer; and each f is now an inversion which wasn't there previously. (Recall that a free-swap subsequently decreases Φ: if we do a free swap, then we didn't really have to add as much to Φ.)

Amortized cost of one MTF search operation:

$$\begin{aligned}
&= \text{search cost} + \text{increased potential}\\
&= K_{\mathrm{MTF}} + (f-b)\\
&= (f+b+1)+(f-b)\\
&= 2f+1\\
&\leq 2(K_{\mathrm{OPT}}-1)+1\\
&= 2K_{\mathrm{OPT}}-1
\end{aligned}$$

Since our request sequence has length m, the cost of m searches is therefore $2\text{comp-cost}_{\mathrm{OPT}}-m$.

□

2.4 Questioning the model: Better than Dynamic Optimality?

The result that MTF is 2-competitive holds in the Sleator-Tarjan cost model, which is unnatural in that reversing a list of n elements costs $\Theta(n^2)$. The Sleator-Tarjan model corresponds to the situation in which we may use exactly one pointer into the list. In a more realistic model in which $O(1)$ pointers into the list are available, reversing the list costs $O(n)$ time[1]. (Swap elements i and $n-i$ as i runs from 1 to $n/2$.) One such model was introduced by J. Ian Munro in [Mun00].

2.5 Order-by-Next-Request (ONR)

Earlier we noted that MTF is 2-competitive in the Sleator-Tarjan cost model. In the slightly less restrictive (and more realistic) Munro model, there is an offline algorithm with an amortized cost of $O(\lg n)$ per access. This algorithm, which appeared in [Mun00], is called Order-by-Next-Request (ONR).

We shall present an illustrative example on which ONR takes $O(\lg n)$ per request while MTF takes $O(n)$ per request. This shows that MTF is not k-competitive, for any k, in the Munro model. Indeed, this argument will apply to any online algorithm, not just MTF.

First, we describe the Munro cost model.

- Each access begins at the beginning of the list.
- Each element visited costs $O(1)$. We must search until the requested element but may search beyond it.
- We may re-order, for free, the part of the list that we have searched. Later we will show that this is reasonable: the re-orderings we do in ONR can be done in time linear in the size of the searched list prefix.

Algorithm. The ONR algorithm is as follows:

- Initially, order the list in the order of anticipated requests.
- When key k is requested, search for the list element with key k; say it is the i'th element in the list. Continue searching until we reach the element numbered $\lceil\!\lceil i \rceil\!\rceil = 2^{\lceil i \rceil}$ (the next power of 2). Reorder the elements we have searched in order of next request (elements to be requested sooner are closer to the front), putting the element just requested first.[2]

Comparing ONR and MTF. Consider a list with n distinct keys; the access sequence consists of a repetition of a particular permutation of the keys. For definiteness, we fix the request sequence consisting of repetitions of $(1, 2, \ldots, 16)$. Table 1 illustrates the execution of ONR on this request sequence. Note that every ith access costs 2^i. The total cost of n accesses for one instance of the permutation is $n \cdot (1/2 \cdot 2 + 1/4 \cdot 4 + 1/8 \cdot 8 + \cdots)$ which is $O(n \lg n)$, giving an amortized cost per access of $O(\lg n)$.

By contrast, MTF (or any online algorithm) might cost $\Theta(n)$ per request, since we can choose a request sequence that always requests the element that is currently last in the list.

[1] Reversing can be done in constant time by keeping a doubly-linked list and a "reverse" bit in each element which, when true, reverses the meaning of forward and backward pointers in subsequent list elements; but if we need to reverse the first half of the list then we still need $O(n)$ time to find the middle element.

[2] We assume that our foreknowledge of request sequence is available in the form of "next request time" for each key at each point in the algorithm, rather than as a linear list of future requests.

2.5.1 How to re-order list prefixes by order of next request for free

In the Munro cost model, we assumed that re-ordering the accessed list prefix by next request is free. To justify this assumption we need to show that our re-ordering can be done in time linear in the length of the prefix. Observe that the prefix to be re-ordered is composed of blocks of size $2^0, 2^1, 2^2 \ldots, 2^{\lceil \lg i \rceil}$ already sorted in ONR (where i is the index of the requested element). For any j, the block of elements at positions $2^j + 1, 2^j + 2, \ldots 2^{j+1} - 1$ is already sorted. We can merge the blocks into a single sorted list, starting from the shortest and merging increasingly longer blocks. Because merging two sorted blocks takes time linear in the length of the merged blocks, all merging operations can be completed in time $2^0 + 2^1 + \cdots + 2^{\lceil \lg i \rceil}$ which is $O(i)$.

Table 1: Execution of the Order-by-Next-Request algorithm. At each point the list is a sequence of blocks, with elements in each block sorted in next-request order.

Search for...	Cost	List after searching (next request circled)															
initialize	16	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	2	2	1	3	4	5	6	7	8	9	10	11	12	13	14	15	16
3	4	3	4	1	2	5	6	7	8	9	10	11	12	13	14	15	16
4	2	4	3	1	2	5	6	7	8	9	10	11	12	13	14	15	16
5	8	5	6	7	8	1	2	3	4	9	10	11	12	13	14	15	16
6	2	6	5	7	8	1	2	3	4	9	10	11	12	13	14	15	16
7	4	7	8	5	6	1	2	3	4	9	10	11	12	13	14	15	16
8	2	8	7	5	6	1	2	3	4	9	10	11	12	13	14	15	16
9	16	9	10	11	12	13	14	15	16	1	2	3	4	5	6	7	8

References

[Alb98] Susanne Albers. Improved randomized on-line algorithms for the list update problem. *Siam Journal on Computing*, 11(3):682–693, June 1998. http://www.informatik.uni-freiburg.de/~salbers/soda95.ps.gz.

[Amb00] Christoph Ambühl. Offline list update is NP-hard. In *Algorithms - ESA 2000, 8th Annual European Symposium, Saarbrücken, Germany, September 5-8, 2000, Proceedings*, volume 1879 of *Lecture Notes in Computer Science*. Springer, 2000. http://www.inf.ethz.ch/personal/ambuehl/publications/ambesa.ps.

[BEY98] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[Mun00] J. Ian Munro. On the competitiveness of linear search. In *Algorithms - ESA 2000, 8th Annual European Symposium, Saarbrücken, Germany, September 5-8, 2000, Proceedings*, volume 1879 of *Lecture Notes in Computer Science*, pages 338 – 345. Springer, 2000. http://db.uwaterloo.ca/~imunro/competlin.ps.

[RWS94] Nick Reingold, Jeffery Westbrook, and Daniel Dominic Sleator. Randomized competitive algorithms for the list update problem. *Algorithmica*, 11(1):15–32, 1994. http://citeseer.nj.nec.com/reingold92randomized.html.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, February 1985. http://doi.acm.org/10.1145/2786.2793.

[Tei93] Boris Teia. A lower bound for randomized list update algorithms. *Information Processing Letters*, 47(1):5–9, 1993.