‘

COMPSCI 590: Optimization and Decision-making under Uncertainty March 24, 2015

Lecture 19

Lecturer: Kamesh Munagala *Scribe: Nat Kell*

1 Introduction

In this lecture, we will examine the online algorithms for the *list update* problem: given a linked list and sequence of requests for elements in the list, how should the list be maintained in order to minimize total search time? We will first show the that the natural approach of ordering based on frequency is not competitive. Next, we show that the algorithm Move to Front (MTF), which moves the most recently requested element to the front of the list, achieves a competitive ratio 2. The analysis of MTF will introduce *potential functions*, which is a widely-used approach for analyzing online algorithms. Finally, we will show that MTF is an optimal online algorithm, i.e., no deterministic algorithm can achieve a competitive ratio better than 2.

2 The List Update Problem

The list update problem is a fundamental online problem that models scenarios where search queries arrive over time for elements stored in a linked list. More formally, the problem is defined as follows: we are given an online sequence of N key access requests $r_1, \ldots, r_N$, where each $r_i \in \{1, \ldots n\}$. The algorithm maintains a linked list L of length n, each element corresponding to one of the n potential keys. When an key access r_i arrives, the algorithm pays a search cost equal to current position of r_i in L (i.e., it has to traverse the linked list to find the position of r_i). We then assume that the algorithm can move r_i to any previous position in the list for free, where the idea is that we might want to make r_i more readily available for future requests. The objective is the minimize the total search cost over all requests in the sequence.

2.1 First approach: Order by Frequency

A natural first attempt for an algorithm is to order the list based on the current access frequency of the keys. Intuitively, this algorithm seems reasonable since if we want to keep our overall search cost low, we want more frequently requested elements to be closer to the front of the list. Also note this ordering can be maintained in the model of the problem, since when the currently requested key can only move up in frequency rank when it is requested.

Unfortunately, the following lemma shows that ordering by frequency is trivially bad with respect to competitive analysis.

Lemma 1. *The competitive ratio of Order by Frequency (Frequency) is* $\Omega(n)$.

Proof. Consider the following request sequence: 1 is requested n times, then 2 is requested n times, and so on, ending with n being requested n times. Consider the cost of Frequency on the the requests for keys $n/2, \ldots, n$ (the last $n^2/2$ requests). Each of these requests will have cost at least $n/2$ since the first $n/2$ requests have frequency n from the first half the request sequence. Therefore, the overall cost of accessing these requests is $\Omega(n^3)$. However, the optimal algorithm has cost $O(n^2)$, since we can just move the currently

requested key to the front of the list (overall, we have n move-to-front operations of cost at most n, and then the remaining $n^2 - n$ request have a cost of 1). Thus, Frequency has competitive ratio $\Omega(n)$. □

3 A competitive algorithm: Move to Front

Examining the proof Lemma 1, the problem with ordering by frequency is that we are ignoring the *locality* of the request sequence (i.e., recently requested items might soon be requested again and so we should keep them close to the front of list). In fact, the optimal algorithm for the above instance is to ignore overall frequency of the elements altogether and always move the currently requested key to the front of the list. This algorithm, which we call *Move to Front* (MTF), turns out to be a constant-competitive online algorithm. In this section, we show MTF achieves a competitive ratio of 2.

3.1 Potential functions

To analyze MTF, we will use a tool known as a *potential function* $\Phi(i)$ that maps the state of the MTF's list after the ith request to a non-negative value, where $\Phi(0)$ is the potential of the starting state. Let $\text{MTF}(i)$ be cost to access request r_i. Instead of analyzing just this actual cost (which is to hard to characterize over all instances), we will instead examine the *amortized cost* of the algorithm, where the amortized cost of the ith request $A(i)$ is defined as:

$$A(i) = \text{MTF}(i) + \Phi(i) - \Phi(i-1). \tag{1}$$

Notice that if we sum amortized cost over all requests, the series telescopes and we are only left with terms that depend on the the actual cost, the initial value of the potential, and the final value of the potential. More formally:

$$\sum_i^N A(i) = \sum_i^N (\text{MTF}(i) + \Phi(i) - \Phi(i-1)) = \sum_i^N \text{MTF}(i) + \Phi(N) - \Phi(0). \tag{2}$$

Observe that $\sum_i^n \text{MTF}(i)$ is just the overall cost of MTF; therefore, if we define the potential function Φ so that $\Phi(n) - \Phi(0)$ is positive, then the overall amortized cost is an upper bound on the cost of the algorithm. Our strategy will be define the potential function so that the change in potential $\Phi(i) - \Phi(i-1)$ is negative when the cost of the algorithm $\text{MTF}(i)$ is high. Clearly, on cheaper requests the potential will have to be positive since the overall contribution of the potential is positive, but hopefully the added is cost small enough that we can obtain a tight bound. Intuitively, we can think of the function as a piggy bank, where on cheap operations it saves up extra money that we can later spend on more expensive iterations.

3.2 Analysis of Move to Front

To define our potential function for our MTF analysis, let $V(i)$ be the set of pairs (x,y) such that x is *inverted* with y in MTF's list with respect to optimal solution's list after iteration i , i.e., after iteration i, x comes before y in MTF's list but y comes before x in the optimal list (or vice versa). We define our potential function to be $\Phi(i) = |V(i)|$. Notice that $\Phi(i)$ is always positive and $\Phi(0)$ is 0 (since the lists for both solutions start off identical), and therefore the amortized cost using this potential will be an upper bound on actual cost.

Using this potential function, we now show the main theorem of the lecture:

Theorem 2. *Let MTF_I and OPT_I be the cost of Move to Front and the optimal solution on instance I, respectively. Then for all instance I, $MTF_I \leq 2OPT_I - N + \binom{n}{2}$. In particular, MTF approaches a competitive ratio of 2 as the size of the request sequence $N \to \infty$.*

Proof. Fix an instance I. Let $\mathrm{OPT}(i)$ be the cost of the optimal solution on request r_i, and recall that $\mathrm{MTF}(i)$ denotes the cost of MTF on request r_i. Observe that $\Phi(N) \leq \binom{n}{2}$ is a generic upper bound on the number of inversions between any two sequences and by definition $\Phi(0) = 0$; therefore, in order to show our desired bound, it suffices to upper bound amortized cost of the ith request as follows:

$$\mathrm{MTF}(i) + \Phi(i) - \Phi(i-1) \leq 2\mathrm{OPT}(i) - 1. \tag{3}$$

To establish Eqn. (3), let k be the number of elements that come before r_i in *both* MTF's list and the optimal list, and let ℓ be the number of elements that come before r_i in MTF's list but come after r_i in the optimal list. Based on these definitions, we have that $\mathrm{MTF}(i) = \ell + k + 1$ and $\mathrm{OPT}(i) \geq k + 1$. Now observe that after r_i is moved in both lists, exactly ℓ inversions are destroyed (since in MTF's list r_i is at the front and now all these ℓ elements come after r_i in both lists) and that at most k new inversions are created. Collecting these observations, we can bound the amortized cost as follows:

$$\begin{aligned}
\mathrm{MTF}(i) + \Phi(i) - \Phi(i-1) &\leq k + \ell + 1 + (k - \ell) \\
&= 2k + 1 \\
&\leq 2(\mathrm{OPT}(i) - 1) + 1 = 2\mathrm{OPT}(i) - 1,
\end{aligned}$$

as desired. □

4 Lower bound

To end the lecture, we will show that MTF is actually an optimal online algorithm, i.e., any online algorithm must be at least 2-competitive.

Theorem 3. *Any online algorithm for the list update problem has competitive ratio at least 2.*

Proof. Fix an algorithm A. The adversary produces an instance of length N as follows. For any time step, there is always some element that is last in the algorithm's current list. The adversary will simply request this element for every time step. Clearly the total cost of the algorithm is Nn since it must traverse the entire list every time. Therefore we just need to argue there exists a solution that achieves cost better than $Nn/2$.

To establish this, observe that maintaining a random permeation of the elements 1 through n has an expected cost of $Nn/2$ for each of the algorithm's requests, since on average any element will be in the middle of the permutation. Thus, there must exist at least one permeation that has actual cost $Nn/2$ on the instance (just by a probabilistic method argument). The optimal solution will simply maintain this permeation for its list, which gives us the desired competitive ratio of 2. □