Advanced Data Structures Jan-Apr 2012

Lecture 2 January 11, 2012

Lecturer: Venkatesh Raman *Scribe: Anil Shukla*

1 Overview

In the last lecture we were studying self organizing lists. We had introduced some self-organizing heuristics such as 'move to front', 'transpose' and 'frequency count'. We had ended that lecture by stating two results, first by Bentley and McGeoch[1] that the cost of 'move to front' heuristic is atmost twice the cost of the static optimal cost. The second result by Sleator and Tarjan[2]: the cost of 'move to front' heuristic for a long sequence of operations is at most twice the cost of any self-adjusting heuristic to serve that sequence that even knows the future access sequence and can perform exchange after every operation (at a cost).

In this lecture we will first show these results after defining the relevent notions precisely. Finally we will start with 'self organizing binary search trees'.

2 Self Organizing Lists

Consider the *dictionary* problem of maintaining a set of items to support the following three operations:

- **access(i)** – Locate item i in the set if exists.
- **insert(i)** – Insert item i in the set if it is already not there.
- **delete(i)** – Delete item i from the set if it is there.

We consider the implementation of this set of operations in a singly linked unsorted list. We are interested in the total time for a sequence of operations.

2.1 Static Offline Optimal Algorithm

Suppose we know the frequency f_i of accessing item $i, 1 \leq i \leq n$. Then one way to minimize the total cost for the sequence of searches is to arrange the list in nonincreasing order of the frequencies, and it is easy to see that this order minimizes the total cost yielding a cost of $\sum_{i=1}^{n} if_i$ where we call i as the item in the i-the location of this list (ordered by freqencies). We call this optimal cost the 'static offline optimal cost' (static because the access operations are not allowed to move the elements) and call such an algorithm static offline optimal algorithm (offline because algorithm knows every thing needed in advance).

2.2 Move-to-Front Heuristic

In this heuristic after accessing and inserting an element, move it to the front of the list without changing the relative order of the other items. We will call algorithm using this heuristic as MTF-algorithm.

2.3 Self Organizing Heuristics

- By 'self organizing' we mean, after any of the operation (access, insert and delete) exchange elements to aid future operation.
- Define the cost of **access(i)**, **delete(i)** to be the number of elements before i and the cost of **insert(i)** to be the number of elements in the list at the time i was inserted.
- Define the cost to be 0 only for moving the accessed element closer to the front, and for all other exchanges (swapping with an adjacent element) define the cost to be 1.

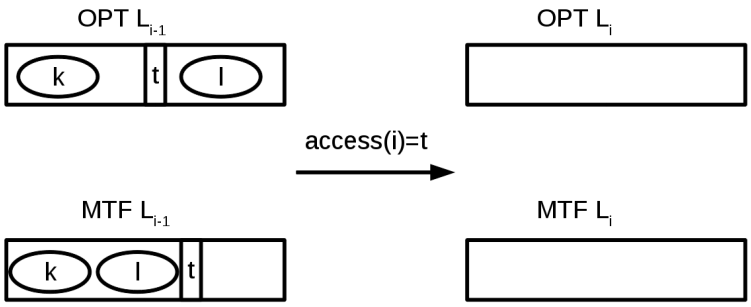
Let n be the maximum number of elements in the list and m be the number of operations. Let $\mathbf{C_A}(\sigma)$ be the total cost of algorithm A on a sequence σ of requests (operations) and $\mathbf{C_{OPT}}(\sigma)$ be the cost for an OPT algorithm on σ. **OPT** is the heuristic which takes minimum cost, it knows the future access sequence and can perform exchanges after every operation in the cost model above (i.e. cost 0 for moving accessed element to the front and cost 1 for any other exchange).

Theorem 1. *[2]* $\mathbf{C_{MTF}}(\sigma) \leq 2\mathbf{C_{OPT}}(\sigma)$.

Proof. Suppose we have a sequence σ of operations. Call the list of elements maintained by **OPT** algorithm after the i^{th} operation as OPT L_i and similarly MTF L_i. Let $\phi(\mathbf{L_j})$ be the potential of the list after the j^{th} operation. We will define it as:

$$\phi(L_j) = \begin{cases} \text{number of inversions between MTF } L_j \text{ and OPT } L_j \text{ or} \\ \\ \text{number of pairs (x,y) such that x appears before y in MTF } L_j \\ \text{ but x appears after y in OPT } L_j + \text{ number of pairs (x,y) such that x is after y} \\ \text{in MTF } L_j \text{ but x is before y in OPT } L_j \end{cases}$$

Let we are accessing element t on the i^{th} operation and assume that t is present in both the lists. Let k be the number of elements in MTF L_{i-1} before t that also appears before t in OPT L_{i-1} and let l be the number of elements in MTF L_{i-1} before t but appears after t in OPT L_{i-1}. See the diagram below.



Clearly actual-costs $\mathbf{C_{MTF}(i) = k + l}$ and $\mathbf{C_{OPT}(i) \geq k}$. We know that Am-cost(i)=act-cost(i)+$\phi(L_i)-\phi(L_{i-1})$.

If **OPT** does not move things after the operation, then l inversions are destroyed and k inversions are created by the move to front that moves the element t to the front of the list (in L_i). Therefore, Am-cost(i)= $k + l + k - l \leq 2k \leq 2C_{OPT}(i)$.

If **OPT** makes a free exchange, then the actual-cost does not change and potential can decrease. Hence above inequality still holds (Am-cost(i) may decrease however $C_{OPT}(i)$ remains the same).

If OPT makes a paid exchange (swapping with just adjancent one), $C_{OPT}(i)$ increases by 1, act-cost (to MTF) does not change. Also the potential can increases by atmost 1 by a new inversion created. Therefore the inequality still holds (as LHS can increase by atmost one and RHS increases by one). Hence now total costs for σ is,

$$\mathbf{C_{MTF}}(\sigma) \leq \sum_{\mathbf{i}} \text{am-cost(i)} + \phi(\text{initial-list}) - \phi(\text{final-list})$$

Since $\phi \geq 0$ we have,

$$\begin{aligned}\mathbf{C_{MTF}}(\sigma) &\leq \sum_{\mathbf{i}} \text{am-cost(i)} + \phi(\text{initial-list}) \\ &\leq \mathbf{2C_{OPT}}(\sigma) + \phi(\text{initial-list})\end{aligned}$$

Usually we start from the empty list. Suppose we started from a nonempty list with n elements. Clearly there can be atmost $\binom{n}{2}$ inversions and we have $\phi(\text{intial-list}) = O(n^2)$, Hence we have,

$$\mathbf{C_{MTF}}(\sigma) \leq \mathbf{2C_{OPT}}(\sigma) + \mathbf{O(n^2)}$$

$O(n^2)$ is an over head for the initial set up which we assume to be constant over a sufficiently large sequence of operations (i.e, $m >> n$).

□

2.3.1 Lower bound for competitive ratio

Consider the request sequence that always requests the last element. Clearly amortised cost of MTF or any online algorithm say A on such sequence of request will be $\Omega(n)$. Now consider an static OPT algorithm (i.e, best algorithm). OPT algorithm will organize the elements in non-increasing order of the frequencies. Clearly if every element is equally likely to be searched, then the average time for a search is $(n + 1)/2$. Thus for any sequence of requests σ, we have,

$$\text{Cost-of-static-OPT}(\sigma) \leq \frac{(n+1)}{2}|\sigma|$$

Thus as a consequence, for any sequence of requests σ, we have,

$$2C_{OPT}(\sigma) - |\sigma| \leq n|\sigma| \leq C_A(\sigma)$$

2.3.2 Example for showing MTF is (more than constant time) better than static optimal and frequency count

Suppose we had inserted elements 1 to n in that order. Consider the sequence σ of operations that accesses element i, $(k+i)$ times, where k is some non-negative integer. Clearly the request sequence σ has length $\sum_{i=1}^{n}(k+i) = nk + O(n^2)$. Frequency count and the static optimal heuristic will keep the elements in the order n to 1, since the non-increasing frequencies of elements are in that order, resulting in a total cost of $\sum_{i=1}^{n} i\big(k+n-(i-1)\big) = \Omega(n^2)$, with an amortized cost of $\Omega(n)$ for large enough k. However, the move to front (MTF) heuristic will bring the element to the front after the first access, resulting in a total cost of $nk + O(n^2)$, with amortized cost of $O(1)$.

Before ending today's lecture, we will briefly discuss 'self organizing binary search trees'. The question here is the same, only instead of lists we have binary search trees.

3 Self Organizing Binary Search Trees

As we know there are many types of binary search trees. **AVL** trees and **red-black** trees are both forms of balanced binary search trees (see chapter on red-black tree in [3]). If we do not plan on modifying a search tree, and we know exactly how often each item will be accessed (i.e their frequencies), we can construct an **optimal binary search tree**, which is a search tree where the average cost of looking up an item (the expected search cost) is minimized. Before we proceed futher let us define optimal binary search tree formally.

3.1 Optimal Binary Search Trees

we are given a sequence $K =< k_1, k_2, \ldots, k_n >$ of n distinct keys in increasing order. For each key k_i, we have a probability p_i that a search will be for k_i. Some search may be for values not in K, and so we have $n+1$ "dummy keys" $d_0, d_1, \ldots, d_n$ representing values not in K. In particular, d_0 represents all values less than k_1, d_n represents all values greater than k_n, and for $i = 1, 2, \ldots, n-1$, the dummy key d_i represents all values between k_i and k_{i+1}. For each dummy key d_i, we have a probability q_i that a search will corresponds to d_i. Clearly,

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

Let we have such a binary search tree $\mathbf{T}$ (each key k_i is an internal node and each dummy key d_i is a leaf), and assume that the actual cost of a search is the number of nodes examined, i.e, the depth of the node found by the search in $\mathbf{T}$, plus 1. Then the expected cost of a search in $\mathbf{T}$ is

$$\begin{aligned} E[\text{search cost in T}] &= \sum_{i=1}^{n}(depth_T(k_i)+1).p_i + \sum_{i=0}^{n}(depth_T(d_i)+1).q_i \\ &= 1 + \sum_{i=1}^{n} depth_T(k_i).p_i + \sum_{i=0}^{n} depth_T(d_i).q_i, \end{aligned}$$

where $depth_T$ denotes a node's depth in the tree T. We call a binary search tree whose expected search cost is smallest an **optimal binary search tree**.

Using dynamic programming one can construct an optimal binary search tree in time $\Theta(n^3)$ (see the chapter on dynamic programming in [3]) . Knuth[4] improves the algorithm to run in $\Theta(n^2)$ time. Improving the runtime to $o(n^2)$ is a long standing open problem.

Even if we only have estimates of the search costs, such a system can considerably speed up lookups on average. For example, if you have a BST of English words used in a spell checker, you might balance the tree based on word frequency that is, placing words like"the" near the root and words like "mycophagist" near the leaves. Such a tree might be compared with Huffman trees, which similarly seek to place frequently-used items near the root inorder to produce a dense information encoding; however, Huffman trees only store data elements in leaves and these elements need not be ordered.

If we do not know the sequence in which the elements in the tree will be accessed in advance, we can use **splay trees** which are asymptotically as good as any static search tree we can construct for any particular sequence of lookup operations. Informally, A **splay tree** is a binary search tree that automatically moves frequently accessed elements nearer to the root.

We will discuss **splay trees** and how they achieve static optimality even without knowing the frequencies in advance next time.

References

[1] J. Bentley and C. C. McGeoch, Amortized analyses of self-organizing sequential search heuristics, *Communications of the ACM* **28**:404-411, 1985.

[2] D. D. Sleator and R. E. Tarjan, Amortized efficiency of list update and paging rules, *Communications of the ACM* **28**: 202-208 (1985).

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, Third edition, Prentice-Hall India Pvt Ltd (2010).

[4] Donald E. Knuth. Optimum binary search trees. *Acta Informatica* **1**: 14-25, 1971.