CS 840 **Fall 2016**

Self Organizing Linear Search

We will consider searching under the comparison model

Binary tree – lg n upper and lower bounds
This also holds in "average case"
Updates also in O(lg n)

Linear search – worst case n
If all elements have equal probability of search, expected time is (n+1)/2 in successful case(n for unsuccessful search)

Real Questions:

1. What if probabilities of access differ? How well can we do under stochastic model? (Easiest example: p_i is probability of requesting element i, probabilities are independent)

2. What if these (independent) probabilities are not given?

3. How do these methods compare with the best we could do?
   - given the frequencies (and later – given the <u>sequence</u> of requests)

   This leads to issues of

   - expected behaviour (given independent probabilities)

   - what if probabilities change

   - amortized behaviour (running versus an adversary so we are considering worst-case but for a <u>sequence</u> of operations)

   - amortized or expected cost could be compared with but possible for the given probabilities/sequence … perhaps compare with optimal static structure

   - consider our structures that will adapt, (self-adjust, though perhaps <u>on-line</u> and compare with the optimal adjusting structure that knows the sequence in advance (off-line)

     Model is a key issue

4. How can we develop self-adjusting data structures? – Adapt to changing probabilities.

Start with linear search. As noted

Worst case n (succ); also amortized
"Average (all equal probabilities) (n+1)/2

– it doesn't matter how you change the structure for worst case or for all probabilities same and independent.

But

1) Start with $\{P_i\}$ $i = 1,n$ $P_i > P_{i+1}$ independent (Stochastic process)

2,3) Clearly the best order is $a_1 \ldots a_n$. Expected cost would be $Sopt = \sum_{i=1}^{n} i\, p_i$

Clearly, we could count accesses and converge
– But count and "chase" probability changes.

4) How can we adapt to changes in probability or "self adjust" to do better? -- other than count

Move elements forward as accessed

- Move to Front MTF
- Transpose TR
- Move halfway ---

Theorem: Under the model in which the probabilities of access are independent, and fixed, the transpose rule performs better than move to front unless

– $n \leq 2$ or
– all p_i's are the same, in which case the rules have the same expected performance (Rivest, *CACM '76*)

Theorem: Under the model in which probabilities of access are independent (and fixed) the move to front gives an expected behaviour asymptotically of $< 2 \Sigma\, i\, p_i$, i.e. less than a factor of 2 of the optimal (Rivest, *CACM '76*).

Can this be improved?

If all probabilities equal MTF = Sopt, the actual result is harder (much).

With $P_i = i^{-2} / (\pi^2/6)$ as $n \to \infty$ ($\pi^2/6$ – fudge factor)

MTF/Sopt = $\pi/2$ (Gonnet, Munro, Suwanda *SICOMP '81*)

and that is the worst case (Chung, Hajela, Seymour *JCSS '88*) (uses Hilbert's inequalities)

But how about – an amortized result, comparing these with Sopt in worst case. Note Sopt takes into account the frequencies

Transpose is a problem: alternate request for last 2 values - n is cost; MTF uses only 2 .

Move to Front – although "disaster" may be better

But how do we handle the "startup"

– note Rivest result asymptotic.

Bentley-McGeough (*CACM* '85).

The model:

Start with empty list
Scan for element requested, if not present,
Insert (and charge) as if found at end of list (then apply heuristic)

Theorem: Under the "insert on first request" startup model, the cost of MTF is $\leq$ 2 Sopt for any sequence of requests.

Proof. Consider an arbitrary list, but focus only on searches for b and for c, and "unsuccessful" comparisons where we compare query value "b or c" versus the other "c or b".

Assume sequence has k b's
and m c's $k \le m$

Sopt order is cb
and there are k "unsuccessful comparisons"

What order of requests maximizes this number under MTF? Clearly

$$c^{m-k}(bc)^k$$

So there are 2k "unssuccessful compares" (one for each b & one for each of the last k c's)

Now observe that this holds in any list for any pair of elements.

Sum over all

$$\text{Cost} \le 2\ \text{Sopt}$$ □

Note: This bound is tight.
Given $a_1, a_2, \ldots, a_n$, repeatedly ask for last in list, so all requested equally often.

Cost is n per search
Whereas "do nothing" = Sopt ≈ (n+1)/2

Note again Transpose is a disaster if we always ask for the last in its list.

Observe, for some sequences we do a lot better than static optimal $a^n{}_1\, a^n{}_2\, a^n{}_3 \ldots a^n{}_n$

Sopt - $(n+1)/2$

MTF - $\left(\frac{n+1}{2} + n - 1\right)/n = \frac{n+1+2n-2}{2n}$
$= \frac{3}{2} - O(1/n)$

Next – Suppose you are given the entire sequence in advance. Sleator & Tarjan (1985).

The model we will discuss may or may not be realistic, but it is a benchmark.
We consider the idea of a "dynamic offline" method. The issue is:

on line: Must respond as requests come
vs
offline: Get to see entire schedule an determine how to move values

$$\text{Competitive Ratio of Alg} \equiv \text{Worst Case of } \frac{\text{Online time of Alg}}{\text{Optimal Offline time}}$$

A method is "competitive" if this ratio is a constant.

But the model becomes <u>extremely</u> important

Basics – Search for or change element in position i: scan to location i at cost i.

Unpaid exchange – can move element i closer to front of list free (keep others in same order)

Paid – can swap adjacent pairs at cost 1
Borodin and El-Yaniv prohibit going past location i
Sleator-Tarjan proof seems ok though.

Issues – Long/Short scan (ie passing location i)
Exchanging only adjacent values

Further

Access costs i if element in position i

Delete costs i if element in position i

Insert costs n+1 if n elements already there.

After any we can apply update.

Theorem: Under the model described, MTF is within a factor of 2 of offline optimal i.e. is 2-competitive.

Let A denote any algorithm, and MF denote the move to front heuristic.

We will consider the situation in which we deal with a sequence, S, having a total of M queries and a maximum of n data values. By convention we start with the empty list. The cost model for a search that ends by finding the element in position i is

i + # paid exchanges

Recall the element sought may be moved forward in the list at no charge (free exchange) while any other moves must be made by exchanging adjacent values.

Notation

$C_A(S)$ = total cost of all operations in S with algorithm A

$X_A(S)$ = # paid exchanges

$F_A(S)$ = # free exchanges

Note: $X_{MF}(S) = X_T(S) = X_{FC}(S) = 0$

(T denotes the transpose heuristic, FC denotes frequency count)

$F_A(S) \leq C_A(S) - M$

(Since after accessing the i^{th} element there are at most i-1 free exchanges)

Theorem: $C_{MF}(S) \le 2C_A(S) + X_A(S) - F_A(S) - M$, for any algorithm A starting with the empty set.

Proof: The key idea is the use of a potential function Φ. We run algorithm A and MF, in parallel, on the same sequence, S.

Φ maps the configuration of the current status of the two methods onto the reals.

Running an operation (or sequence) maps Φ to Φ' and the amortized time of the operation is

$$T + \Phi' - \Phi$$

(i.e. amortized time = real time + $\Delta\Phi$)
(so we will aim at amortized time as an overestimate)
The j^{th} operation takes actual time (cost) t_j and amortized cost a_j

$$\Sigma_j t_j = \Phi - \Phi' + \Sigma_j a_j$$

where Φ is the initial potential and Φ', the final.

Φ is defined as the number of inversions between the status of A and MF, at the given time.

So $\Phi \le n (n-1) / 2$

We want to prove that the amortized time to access element i in A's list is at most 2i - 1 in MF's.

Similarly inserting in position i+1 has amortized cost 2(i+1) – 1. (Deletions are similar).

Furthermore: we can make the amortized time charged to MF when A does an exchange

-1 for free exchanges
at most +1 for paid exchanges

Initial configuration – empty; so $\Phi = 0$

Final value of Φ is nonnegative

So actual MF cost $\le$ Σ amortized time $\le$ our bound

{ access or insertion amortized time $\le$ $2C_A - 1$;
amortized delete time $\le 2C_A - 1$.
The -1's, one per operation, sum to -M }

Now we must bound the amortized times of operations.

Consider access by A to position $\underline{i}$ and assume we go to position $\underline{k}$ in MF.

$$x_i \quad = \quad \text{\# items preceding } \underline{\text{it}} \text{ in MF, but not in A}$$

$$\text{so} \quad \text{\# items preceding } \underline{\text{it}} \text{ in both is } (k - 1 - x_i)$$

Moving $\underline{\text{it}}$ to front in MF creates

$$k - 1 - x_i \text{ inversions}$$

and destroys x_i others

so amortized time is

$$k + (k - 1 - x_i) - x_i = 2(k - x_i) - 1$$

But $(k - x_i) < i$ as $k-1$ items precede $\underline{\text{it}}$ in MF and only $i-1$ in A.

So amortized time $\leq$ $\underline{2i - 1.}$

The same argument goes through for insert and delete.

An exchange by A has zero cost to MF,
so amortized time of an exchange is just increase in # inversions caused by exchange,
i.e. 1 for paid, -1 for free.

Extension

Let MF(d) $(d \geq 1)$ be a rule by which the element inserted or accessed in position k is moved at least $k/d - 1$ units closer to the front. Then

$$C_{MF(d)}(S) \leq d\,(2C_A(S) + X_A(S) - F_A(S) - M)$$

{ eg. MF $\equiv$ MF(1) }

Also

Theorem: Given any algorithm A running on a sequence S, there exists another algorithm for S that is no more expensive and does no paid exchanges.

Proof Sketch: Move elements only after an access, to corresponding position. There are details to work through.

Further applications can be made to paging.

However – there is a question about the model.

Given the sequence

$$1 \rightarrow 2 \rightarrow \bullet\bullet\bullet \rightarrow n/2 \rightarrow n/2+1 \rightarrow \bullet\bullet\bullet \rightarrow n$$

suppose we want to convert it to

$$n/2 \rightarrow \bullet\bullet\bullet \rightarrow n \rightarrow 1 \rightarrow \bullet\bullet\bullet\ n/2$$

what "should" I pay?
Observe that all only 3 pointers are changed, though presumably I have to scan the list ($\Theta(n)$).

Sleator and Tarjan model says $\Theta(n^2)$, perhaps $\Theta(n)$ is a fairer charge.

So for an offline algorithm: To search for element in position i, we probe up to position k ($k \geq i$) and can reorder elements in positions 1 through k. Cost is k.

J.I. Munro: On the Competitiveness of Linear Search. ESA '00 (LNCS 1879 pp 338-345)

Consider the following rule, Order by Next Request (ONR):

To search for element in position i, continue scan to position $2^{\lceil \lg i \rceil}$.
Then reorder these $2^{\lceil \lg i \rceil}$ elements according to the time until their next requests.
(The next of these to be accessed goes in position 1)

$$\text{Cost } 2^{\lceil \lg i \rceil} = \Theta(i)$$

How does this do? First try a permutation, say 1, …, n (let n = 16) and assume 1 is initially in the last half).
(To simplify diagram we move requested value to front)

Request	Cost	New Ordering
1	16	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
2	2	2 1 • •
3	4	3 4 1 2 • •
4	2	4 3 • •
5	8	5 6 7 8 1 2 3 4 • •
6	2	6 5 • •
7	4	7 8 5 6 • •
8	2	8 7 • •
9	16	9 10 11 12 13 13 14 15 16 1 2 3 4 5 6 7 8

Clearly the same cost applies to any permutation

Under our model this cost is ~ n lg n.