CS 624
Lecture 11: Self-Organizing Lists

1 The problem

We have a list L of n elements. The list has a *list head*, sometimes denoted simply by L, which points to the first element of the list. We say the the *rank* of an element x in the list L is

$$\operatorname{rank}_L(x) = \text{distance of } x \text{ from the head of the list } L$$

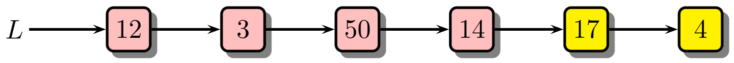
So in particular, we have $1 \le \operatorname{rank}_L(x) \le n$.

We also have two operations:

Access The cost of ACCESS(x) is $\operatorname{rank}_L(x)$.

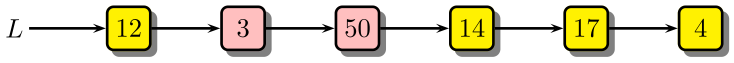
Transpose The cost of transposing two adjacent elements is 1.

Thus in this list



accessing the element with key 14 has a cost of 4.

And transposing 3 and 50 costs 1.



The elements of the list are fixed, but we can rearrange the list whenever we want to by using any number of TRANSPOSE operations.

The idea is that this list is used as part of an algorithm, say, or it is just being used as a lookup table by actual users. In any case, successive ACCESS operations come in, one after the other. Each time an ACCESS operation comes in, we access the element, and then we perform some (possibly 0) TRANSPOSE operations to restructure the list. The idea is that we want to keep the list restructured so that the total cost of all the operations is as small as possible.

A list that is managed in this way is called a *self-organizing* list.

There are two important variants of this problem:

On-line problem: We say this is an *on-line* problem if we can't see ahead—that is, we only see the current ACCESS operation, but we have no knowledge of any later ACCESS operations.

This is similar to the game Tetris, in which you only see the current shape, but you don't know what the next shape will be. (Well, maybe you can sometimes see the next shape, but you certainly can't see all of them.) You just have to do the best you can with the limited information you have.

Off-line problem: We say this is an *off-line* problem if we know at the start the entire sequence S of ACCESS operations.

Clearly this is a lot more information, and so one would expect that we could find the absolutely best sequence of TRANSPOSE operations to use to minimize the total cost.

In fact, this is true. Without giving an explicit algorithm for this, we know that an optimal sequence of operations exists. We call this optimal algorithm OPT, and the total cost of this sequence we call $C_{\mathrm{OPT}}(S)$ (that is, "the cost of the optimal algorithm applied to the sequence S of input operations").

Sometimes you hear off-line algorithms such as OPT referred to as "God's algorithm". I hope no one will be offended by this.

In general, given any algorithm A (either for the on-line or off-line problem), we say that the total cost for processing the sequence S of input access operations is $C_A(S)$.

2 Worst-case analysis for any on-line algorithm

Now we can give a worst-case analysis of any on-line algorithm right away: since we never know what the next ACCESS is going to be, we have to assume it is the worst possible one—that is, that it is an ACCESS of the element that will be at the end of the list after we have finished our current processing. Therefore the cost of each ACCESS will be n (the length of the list L), and so in the worst case, for any on-line algorithm A,

$$C_A(S) = \Omega\big(|S| \cdot n\big)$$

(We are using a common notation here: if X is any set, then $|X|$ denotes the number of elements in X.) Of course the cost could be greater than $|S| \cdot n$ if we performed any transposes. But at any rate, the cost is actually *at least* $|S| \cdot n$.

3 Average-case analysis for any on-line algorithm in a special case

An average-case analysis in general is going to be difficult. But we can consider a simplification consisting of two parts:

1. We start out with the knowledge of the frequency with which each element is accessed. That is, there is a function p which takes each key k and returns the probability with which that key is accessed. So we have

   $$0 \le p(x) \le 1 \qquad \text{for all } x \in L$$

   and

   $$\sum_{x \in L} p(x) = 1$$

   This is actually a pretty severe simplification. There is no guarantee that such a probability distribution exists in practice. But at least we can get some insight into the problem by making this assumption.

2. We use this information to put the list in some order to begin with, and we never change that order during the processing of the algorithm.

If we start off with a certain ordering, then the expected cost will be

$$E\big[C_A(S)\big] = \sum_{x \in L} p(x)\,\mathrm{rank}_L(x)$$

3.1 Exercise *Show that the expected cost (in this version of the problem) is minimized when the list is sorted in decreasing order with respect to p.*

3.2 Exercise *Show that not every sequence of choices leads to a frequency for each element. To do this, of course, you need some sort of counterexample. So you need to show how to construct an infinite sequence of choices such that there is no limiting frequency for at least one element.*

This needs to be done with more than hand-waving. You actually need to write down some specific rules and formulas.

4 The move-to-front algorithm

Based on the average-case analysis (for the special case of the problem) considered above, we might think of something like this:

> **Heuristic:** Keep a count of the number of times each element is accessed, and maintain L in order of decreasing count.

The interesting thing about this is that if there really was a probability distribution for these Access operations on different keys, then the cost to maintain L in order of decreasing count would diminish as time went on, and so this would really be an effective algorithm.

On the other hand, if there really were no probability distribution, then the order of the list would never stabilize.

In practice, what people really found really worked pretty well was an algorithm called Move-To-Front (often simply abbreviated MTF):

```
MOVE-TO-FRONT($L, x$)
    After accessing $x$, move $x$ to the front of the list using TRANSPOSE operations.
```

The cost for the i^{th} operation (accessing element x, say) is then

$$c_i = 2 \cdot \operatorname{rank}_L(x) - 1 \leq 2 \cdot \operatorname{rank}_L(x)$$

Why is this? Well, it costs $\operatorname{rank}_L(x)$ to ACCESS(x). Then it takes $\operatorname{rank}_L(x) - 1$ TRANSPOSE operations to move x to the front of the list.

This is a bit different than the heuristic stated above: it doesn't move x to the front slowly—it moves it all the way at once. One might think that this could be counterproductive. For instance, maybe x was only accessed infrequently. Moving it all the way to the front each time it was accessed might really not be the best thing to do.

On the other hand, this algorithm tends to work well in practice because access requests tend to exhibit what is called *locality*: if an element is accessed, it is likely that the same element will be accessed soon again. So moving that element to the front of the list (and letting it "die off gradually") actually makes a lot of sense in this situation.

But really these arguments are just intuitive, and they are not really conclusive at all. Some partial rigorous results were obtained, but a satisfactory theory of why this algorithm worked so well in practice was missing.

5 The paper of Sleator and Tarjan

All this changed with a remarkable paper of Sleator and Tarjan:

> Daniel D. Sleator and Robert E. Tarjan. *Amortized Efficiency of List Update Rules.* Proceedings of the sixteenth annual ACM symposium on Theory of Computing (STOC '84). December, 1984, pp. 488–492.

Sleator and Tarjan made the following definition:

Definition *An on-line algorithm A is α-competitive if there exists a constant k such that for any sequence S of operations,*

$$C_A(S) \leq \alpha \cdot C_{OPT}(S) + k$$

where OPT is the optimal off-line algorithm.

Note that *there is no assumption that a probability distribution exists.* This property has to work for *all* input sequences (with the same α and k). This is a very strong property. It's not even intuitively clear that it should ever hold.

They then were able to prove this remarkable theorem:

5.1 Theorem (Sleator and Tarjan) MOVE-TO-FRONT *is 4-competitive for self-organizing lists.*

Look at what this says. Suppose for instance that we have the worst case: the sequence always accesses the last element in the list. As we've seen, MOVE-TO-FRONT doesn't do well in this case. *This theorem says that "God couldn't do much better", anyway not better than a factor of 4.* And further, the theorem states that if the sequence of operations exhibits any locality, or anything at all that could be taken advantage of by an optimal off-line algorithm, MOVE-TO-FRONT will take advantage of it as well.

So this is a stunning result.

Note that this is essentially a theorem about amortized costs. That is, we're not concerned with the cost of any specific step—rather, we are concerned with the cost of the first n steps. That's what we mean by amortized analysis.

Further, we don't know what those individual steps are. So this is a much more sophisticated problem than the two we dealt with in the last lecture. And in particular, we can't use the method of "aggregate analysis" here—that method only makes sense when we really know what each step is, so we can add up the costs explicitly.

Therefore, if we are going to analyse this using the methods of amortized analysis, we will of necessity end up either using the accounting method or the potential method. Sleator and Tarjan used the potential method, and we will follow their reasoning here.

To prove this theorem, let us set up some notation: Suppose the i^{th} operation accesses x. Let us set

$$\begin{aligned}
L_i &= \text{MTF's list after the } i^{\text{th}} \text{ access} \\
L_i^* &= \text{OPT's list after the } i^{\text{th}} \text{ access} \\
t_i &= \text{the number of } \textsc{Transpose} \text{ operations that OPT performs at the } i^{\text{th}} \text{ step} \\
c_i &= \text{MTF's cost for the } i^{\text{th}} \text{ operation} \\
&= 2 \cdot \operatorname{rank}_{L_{i-1}}(x) - 1 \\
&\le 2 \cdot \operatorname{rank}_{L_{i-1}}(x) \\
c_i^* &= \text{OPT's cost for the } i^{\text{th}} \text{ operation} \\
&= \operatorname{rank}_{L_{i-1}^*}(x) + t_i
\end{aligned}$$

Now then Sleator and Tarjan used the method of amortized analysis. They first defined an *inversion*. Let us say that $x \prec_{L_i} y$ if x comes before y in the list L_i, and similarly for $x \prec_{L_i^*} y$.

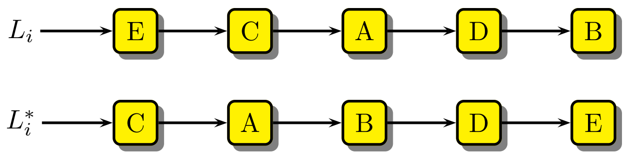
Definition *An inversion at step i is a pair (x, y) of elements of L such that both*

$$\begin{aligned}
x &\prec_{L_i} y \\
y &\prec_{L_i^*} x
\end{aligned}$$

They then defined a potential function $\Phi : \{L_i\} \to \mathbf{R}$ by

$$\Phi(L_i) = 2 \cdot \langle \text{the number of inversions at step } i \rangle$$

For example, if at step i we have



then the inversions are

$$\{(E, C), (E, A), (E, D), (E, B), (D, B)\}$$

so in this case we would have $\Phi(L_i) = 2 \cdot 5 = 10$.

(You might ask: why the factor of 2? This is just to make everything work out correctly later, as we'll see.)

Note that

- $\Phi(L_i) \geq 0$ for all i.
- $\Phi(L_0) = 0$ if MTF and OPT start with the same list.

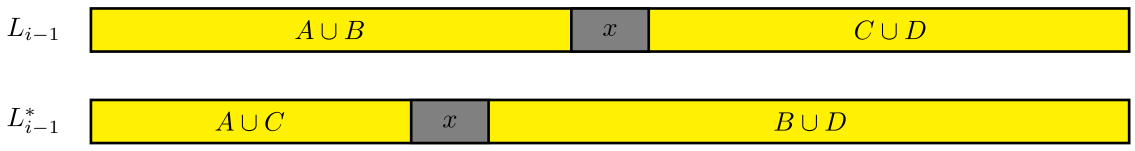
Now let's see how Φ changes with different operations:

First, it's easy to see that a transpose either creates or destroys exactly 1 inversion. Therefore the change in Φ due to a single transpose is ± 2.

To go farther, we need some notation and some pictures. Suppose that operation i accesses element x. Let us set

$$\begin{aligned}
A &= \left\{ y \in L_{i-1} : y \prec_{L_{i-1}} x \text{ and } y \prec_{L^*_{i-1}} x \right\} \\
B &= \left\{ y \in L_{i-1} : y \prec_{L_{i-1}} x \text{ and } y \succ_{L^*_{i-1}} x \right\} \\
C &= \left\{ y \in L_{i-1} : y \succ_{L_{i-1}} x \text{ and } y \prec_{L^*_{i-1}} x \right\} \\
D &= \left\{ y \in L_{i-1} : y \succ_{L_{i-1}} x \text{ and } y \succ_{L^*_{i-1}} x \right\}
\end{aligned}$$

So our two lists L_{i-1} and L^*_{i-1} look like this:



For convenience, let us set

$$\begin{aligned} r &= \operatorname{rank}_{L_{i-1}}(x) = |A| + |B| + 1 \\ r^* &= \operatorname{rank}_{L^*_{i-1}}(x) = |A| + |C| + 1 \end{aligned}$$

Now when MOVE-TO-FRONT moves x to the front, it creates $|A|$ inversions and destroys $|B|$ inversions. And each of the TRANSPOSE operations performed by OPT creates ≤ 1 inversion. Thus, we have

$$\Phi(L_i) - \Phi(L_{i-1}) \leq 2\big(|A| - |B| + t_i\big)$$

Now the amortized cost for the i^{th} operation of MOVE-TO-FRONT with respect to the potential function Φ is by definition

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(L_i) - \Phi(L_{i-1}) \\ &\leq 2r + 2\big(|A| - |B| + t_i\big) \\ &= 2r + 2\Big(|A| - \big(r - 1 - |A|\big) + t_i\Big) && \text{(since } r = |A| + |B| + 1\text{)} \\ &= 2r + 4|A| - 2r + 2 + 2t_i \\ &= 4|A| + 2 + 2t_i \\ &\leq 4|A| + 4 + 4t_i \\ &\leq 4(r^* + t_i) && \text{(since } r^* = |A| + |C| + 1 \geq |A| + 1\text{)} \\ &= 4c_i^* \end{aligned}$$

And now we're just about done. We have

$$C_{\mathrm{MTF}}(S) = \sum_{i=1}^{|S|} c_i \leq \sum_{i=1}^{|S|} \hat{c}_i \leq 4 \sum_{i=1}^{|S|} c_i^* = 4C_{\mathrm{OPT}}(S)$$

And that's it!