Lecture 12

Competitive Analysis

Supplemental reading in CLRS: None

12.1 Online and Offline Algorithms

For some algorithmic problems, the optimal response to a sequence of calls to a procedure depends not just on each input separately, but on the sequence of inputs as a whole. This is analogous to amortized analysis, in which the best running time analysis required us to consider the running time of an entire sequence of operations. When an algorithm expects to be given the entire sequence of inputs ahead of time, it is called an **offline algorithm**; when the algorithm is expected to give an answer after each individual input without knowledge of future inputs, it is called an **online algorithm**.

Example. Traders are faced with the following algorithmic problem: given today's stock prices (and the history of stock prices in the past), decide which stocks to buy and which stocks to sell. Obviously, this is an online problem for everybody. An attempt to treat the trading problem as an offline algorithmic problem is called *insider trading*.

Example. *Tetris* is an online game. The offline version of Tetris would be akin to a jigsaw puzzle, and would not require quick thinking. Of course, we would expect players of the offline game to produce much better packings than players of the online game.

Let's lay out a general setup for analyzing online and offline algorithms. Assume we are given an algorithm A, along with a notion of "cost" C_A. That is, if S is a sequence of inputs, then $C_A(S)$ is the cost of trusting A's response to the inputs in S. For example, A might be a stock broker and $C_A(S)$ might be the net profit after a sequence of days on the stock market. In that case, clients will probably try to learn as much as the can about the function C_A (for example, by observing its past outputs) when deciding whether to hire the stock broker.

In general, no online algorithm will perform optimally on all sequences of inputs—in other words, the so-called "**God's algorithm**" (which performs optimally on every sequence of inputs) will usually be impossible to describe in an online way. This is obvious in the stock market example: if you chose to buy today, what if prices drop tomorrow? If you chose to sell today, what if prices go up tomorrow?

Although an online algorithm may not ever be able to match the performance of God's algorithm, it may be possible for an online algorithm to perform *almost* as well as God's algorithm on every input.

Definition. An online algorithm A is said to be α-**competitive**[1] (where α is a positive constant) if there exists a constant k such that, for every sequence S of inputs, we have

$$C_A(S) \le \alpha \cdot C_{\mathrm{OPT}}(S) + k,$$

where OPT is the optimal offline algorithm, A.K.A. God's algorithm.

12.2 Example: A Self-Organizing List

Problem 12.1. Design a data structure L representing a list of n key–value pairs (with distinct keys), satisfying the following constraints:

- The key–value pairs are stored in a linked list

- There is only one supported operation: ACCESS(x), which returns the element with key x. (It is assumed that the input x is always one of the keys which occurs in L.) The implementation of ACCESS(x) has two phases:

  1. Walk through the list until you find the element with key x. The cost of this phase is $\mathrm{rank}_L(x)$, the rank of the element with key x.

  2. Reorder the list by making a sequence of adjacent transpositions[2] (in this lecture, the word "transposition" will always refer to adjacent transpositions). The cost of each transposition is 1; the sequence of transpositions performed is up to you, and may be empty.

Try to choose a sequence of transpositions which optimizes the performance of ACCESS, i.e., for which ACCESS is α-competitive where α is minimal among all possible online algorithms.

12.2.1 Worst-case inputs

No matter how we define our algorithm, an adversary (who knows what algorithm we are using) can always ask for the last element in our list, so that phase 1 always costs n, and the cost of a sequence S of such costly operations is at least

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

even disregarding the cost of any reordering we do.

12.2.2 Heuristic: Random inputs

As a heuristic (which will seem particularly well-chosen in retrospect), let's suppose that our inputs are random rather than worst-case. Namely, suppose the inputs are independent random variables, where the key x has probability $p(x)$ of being chosen. It should be believable (and you can prove if

---

[1] This is not the only notion of "competitive" that people might care about. We could also consider additive competitiveness, which is a refinement of the notion considered here: We say an algorithm A is *k-additively competitive* if there exists a constant k such that $C_A(S) \le C_{\mathrm{OPT}}(S) + k$ for every sequence S of inputs. Thus, every k-additively competitive algorithm is 1-competitive in the sense defined above.

[2] An *adjacent transposition* is the act of switching two adjacent entries in the list. For example, transposing the elements B and C in the list $\langle A,B,C,D \rangle$ results in the list $\langle A,C,B,D \rangle$.

you wish) that, when $|S|$ is much larger than n, the optimal expected cost is achieved by immediately sorting L in decreasing order of $p(x)$, and keeping that order for the remainder of the operations.[3] In that case, the expected cost of a sequence S of ACCESS operations is

$$\mathbb{E}_{S}[C_A(S)] = \sum_{x \in L} p(x) \cdot \text{rank}_P\left(p(x)\right) \cdot |S|,$$

where $P = \{p(y) : y \in L\}$.

In practice, we won't be given the distribution p ahead of time, but we can still estimate what p would be if it existed. We can keep track of the number of times each key appears as an input, and maintain the list in decreasing order of frequency. This *counting heuristic* has its merits, but we will be interested in another heuristic: the **move-to-front** (MTF) heuristic, which works as follows:

- After accessing an item x, move x to the head of the list. The total cost of this operation is $2 \cdot \text{rank}_L(x) - 1$:

    - $\text{rank}_L(x)$ to find x in the list
    - $\text{rank}_L(x) - 1$ to perform the transpositions.[4]

To show that the move-to-front heuristic is effective, we will perform a **competitive analysis**.

Proposition 12.2. *The MTF heuristic is 4-competitive for self-organizing lists.*

As to the original problem, we are not making any assertion that 4 is minimal; for all we know, there could exist α-competitive heuristics for $\alpha < 4$.[5]

Proof.

- Let L_i be MTF's list after the ith access.
- Let L_i^* be OPT's list after the ith access.
- Let c_i be MTF's cost for the ith operation. Thus $c_i = 2 \text{rank}_{L_{i-1}}(x) - 1$.
- Let c_i^* be OPT's cost for the ith operation. Thus $c_i^* = \text{rank}_{L_{i-1}^*}(x) + t_i$, where t_i is the number of transpositions performed by OPT during the ith operation.

We will do an amortized analysis. Define the potential function[6,7]

$$\Phi(L_i) = 2 \cdot \left(\text{\# of inversions between } L_i \text{ and } L_i^*\right)$$

---

[3] The intuition is that the most likely inputs should be made as easy as possible to handle, perhaps at the cost of making some less likely inputs more expensive.

[4] The smallest number of transpositions needed to move x to the head of the list is $\text{rank}_L(x) - 1$. These $\text{rank}_L(x) - 1$ transpositions can be done in exactly one way. Once they are finished, the relative ordering of the elements other than x is unchanged. (If you are skeptical of these claims, try proving them.) For example, to move D to the front of the list $\langle A, B, C, D, E \rangle$, we can perform the transpositions $D \leftrightarrow C$, $D \leftrightarrow B$, and $D \leftrightarrow A$, ultimately resulting in the list $\langle D, A, B, C, E \rangle$.

[5] The question of whether $\alpha = 4$ is minimal in this example is not of much practical importance, since the costs used in this example are not very realistic. For example, in an actual linked list, it would be easy to move the nth entry to the head in $O(1)$ time (assuming you already know the location of both entries), whereas the present analysis gives it cost $\Theta(n)$. Nevertheless, it is instructive to study this example as a first example of competitive analysis.

[6] The notation $x <_{L_i} y$ means $\text{rank}_{L_i}(x) < \text{rank}_{L_i}(y)$. In other words, we are using L_i to define an order $<_{L_i}$ on the keys, and likewise for L_i^*.

[7] Technically, this is an abuse of notation. The truth is that $\Phi(L_i)$ depends not just on the list L_i, but also on L_0 and i (since L_i^* is gotten by allowing OPT to perform i operations starting with L_0). We will stick with the notation $\Phi(L_i)$, but if you like, you can think of Φ as a function on "list histories" rather than just lists.
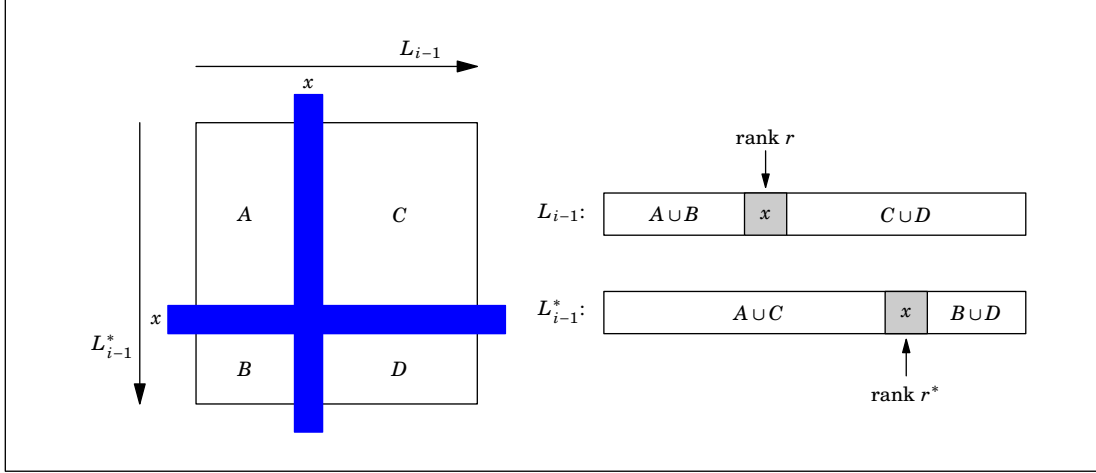
Figure 12.1. The keys in L_{i-1} other than x fall into four categories, as described in the proof of Proposition 12.2.

$$= 2 \cdot \left| \left\{ \text{unordered pairs } \{x, y\} : \ x <_{L_i} y \text{ and } y <_{L_i^*} x \right\} \right|.$$

For example, if $L_i = \langle E, C, A, D, B \rangle$ and $L_i^* = \langle C, A, B, D, E \rangle$, then $\Phi(L_i) = 10$, seeing as there are 5 inversions: $\{E, C\}$, $\{E, A\}$, $\{E, D\}$, $\{E, B\}$, and $\{D, B\}$. Note the following properties of Φ:

- $\Phi(L_i) \geq 0$ always, since the smallest possible number of inversions is zero.
- $\Phi(L_0) = 0$ if MTF and OPT start with the same list.
- A transposition either creates exactly 1 inversion or destroys exactly 1 inversion (namely, transposing $x \leftrightarrow y$ toggles whether $\{x, y\}$ is an inversion), so each transposition changes Φ by $\Delta\Phi = \pm 2$.

Let us focus our attention on a single operation, say the ith operation. The keys of L fall into four categories (see Figure 12.1):

- A: elements before x in L_{i-1} and L_{i-1}^*
- B: elements before x in L_{i-1} but after x in L_{i-1}^*
- C: elements after x in L_{i-1} but before x in L_{i-1}^*
- D: elements after x in L_{i-1} and L_{i-1}^*.

Let $r = \text{rank}_{L_{i-1}}(x)$ and $r^* = \text{rank}_{L_{i-1}^*}(x)$. Thus

$$r = |A| + |B| + 1 \quad \text{and} \quad r^* = |A| + |C| + 1.$$

Now, if we hold L_{i-1}^* fixed and pass from L_{i-1} to L_i by moving x to the head, we create exactly $|A|$ inversions (namely, the inversions $\{a, x\}$ for each $a \in A$) and destroy exactly $|B|$ inversions (namely, the inversions $\{b, x\}$ for each $b \in B$). Next, if we hold L_i fixed and pass from L_{i-1}^* to L_i^* by making whatever set of transpositions OPT chooses to make, we create at most t_i inversions (where t_i is the number of transpositions performed), since each transposition creates at most one inversion. Thus

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

The amortized cost of the ith insertion of MTF with respect to the potential function is therefore

$$\widehat{c}_i = c_i + \Phi(L_i) - \Phi(L_{i-1})$$

$$\leq 2r + 2\left(|A| - |B| + t_i\right)$$
$$= 2r + 2\left(|A| - (r - 1 - |A|) + t_i\right)$$
$$= 4|A| + 2 + 2t_i$$
$$\leq 4\left|r^* + t_i\right| \qquad (\text{since } r^* = |A| + |C| + 1 \geq |A| + 1)$$
$$= 4c_i^*.$$

Thus the total cost of a sequence S of operations using the MTF heuristic is

$$C_{\text{MTF}}(S) = \sum_{i=1}^{|S|} c_i$$

$$= \sum_{i=1}^{|S|} \left(\widehat{c}_i + \Phi(L_{i-1}) - \Phi(L_i)\right)$$

$$\leq \left(\sum_{i=1}^{|S|} 4c_i^*\right) + \underbrace{\Phi(L_0)}_{0} - \underbrace{\Phi\left(L_{|S|}\right)}_{\geq 0}$$

$$\leq 4 \cdot C_{\text{OPT}}(S).$$

$\square$

Note:

- We never found the optimal algorithm, yet we still successfully argued about how MTF competed with it.
- If we decrease the cost of a transposition to 0, then MTF becomes 2-competitive.
- If we start MTF and OPT with different lists (i.e., $L_0 \neq L_0^*$), then $\Phi(L_0)$ might be as large as $\Theta(n^2)$, since it could take $\Theta(n^2)$ transpositions to perform an arbitrary permutation on L. However, this does not affect α for purposes of our competitive analysis, which treats n as a fixed constant and analyzes the asymptotic cost as $|S| \to \infty$; the MTF heuristic is still 4-competitive under this analysis.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2012