COS 340: Reasoning About Computation* Online Algorithms 1

Moses Charikar

March 9, 2014

1 Introduction

In this lecture, we will introduce the notion of online algorithms and competitive analysis. We will study the Rent or Buy problem, and then discuss the List Update problem.

In many situations, algorithms must make decisions without complete knowledge of the entire input. An important class of such algorithms is *online algorithms* which receive their input as a sequence of requests and are required to make decisions based on the requests seen so far without any knowledge of future requests. Further, decisions made early on cannot be undone when later requests are revealed. Online algorithms should be contrasted with *offline algorithms* which do have the entire input available to them – this is just the usual notion of an algorithm. [1]

The study of online algorithms is all about decision making in the face of uncertainty. How should the algorithm make decisions without knowledge of future requests ? How should we evaluate the quality of such an algorithm ? I should mention that people do study scenarios where we have partial information about the future – for example, sometimes inputs are assumed to come from a probability distribution. In online algorithms, we take a worst-case analysis viewpoint, and assume that we know nothing about future requests. While we haven't yet discussed a measure of goodness for online algorithms, our goal is to design algorithms that work well for all inputs.

2 Rent or Buy

In order to motivate the performance measure that is used to evaluate the quality of an online algorithm, let's introduce an example which we will discuss in some detail in this

*Princeton University, Spring 2014

[1] You may wonder about the choice of name for *online algorithms*. Given other connotations of the word *online*, perhaps a more informative name would be *real-time algorithms*, but that term is already taken and has a different meaning in Computer Science. In any case, when the term *online algorithm* was coined, the word *online* did not have the meaning attached to it that it has today.

lecture:

Alice[2] is a student in COS 340 and a budding photographer. She decides to develop her hobby and earn some extra money on the side photographing events and doing photo shoots in Princeton. Of course, she has to upgrade her amateur equipment and needs a good camera and lens to get started. Let's say that the cost of buying the equipment is \$1000 (most likely an underestimate, but let's stick with a nice round number for our example). She also has the option of renting a camera and lens and the cost of this is a much more affordable \$100 per assignment. Having started her new photography business, she is now faced with the following Rent or Buy problem: each time she gets a new assignment, should she rent the camera for the assignment or buy it? We'll assume in our example that once she buys the equipment, it is good for all future assignments.

What strategy should Alice use for the online problem she faces ? In order to motivate the performance measure for online algorithms, let's examine two extreme strategies:

1. She could decide to buy the camera when she gets her first assignment. The trouble with making an expensive investment upfront is that she is not sure how many photo assignments she will actually get. If she is unlucky and ends up with only one assignment, then buying the equipment would be a waste of money in retrospect (let's ignore the resale value in our example).

2. She could decide to rent always and never buy the camera. The trouble with this strategy is that if she gets a very large number of assignments, she will run up a large bill with her rental costs. In retrospect, she ought to have bought the camera and saved on her rental expenses.

Note that in both cases, we compare the cost of Alice's strategy to the best strategy in retrospect, i.e. the best strategy had we known how many assignments Alice would eventually get. Informally, we would like to design a strategy that does well compared to the best strategy in hindsight. This is the motivation behind the *competitive ratio* measure for online algorithms which we now define.

Let $\sigma = (\sigma_1, \sigma_2, \ldots)$ be the request sequence. Let $C_A(\sigma)$ denote the cost of online algorithm A on σ. Let $C_{OPT}(\sigma)$ denote the cost of the optimal solution on σ. Note that the optimal solution is computed with knowledge of the entire input sequence. We say that algorithm A is c-competitive if for all request sequences σ, $C_A(\sigma) \leq c \cdot C_{OPT}(\sigma)$ (in other words, the algorithm A pays no more than a factor of c times the optimal cost for the request sequence). In fact, this also means that for all points in the request sequence, the algorithm A pays no more than a factor of c times the optimal cost for the portion of the request sequence seen so far. The *competitive ratio* of the algorithm is the worst case over all request sequences of the ratio $C_A(\sigma)/C_{OPT}(\sigma)$. So when we say that A is c-competitive, we mean that the competitive ratio is at most c.

[2]In case you haven't realized this already, **A**lice and **B**ob are the two most frequently encountered characters in theoretical computer science. In fact, if you take a cryptography class, they will probably pop up in every lecture.

Now we turn to analyzing the competitive ratio for Alice's RENT OR BUY problem. Consider the following strategy: Suppose Alice rents a camera for the first 9 assignments and buys a camera if she gets a 10th assignment. Let's call this strategy A_9 (since we rent a maximum of 9 times). Later, we will consider generalizations of this strategy where we rent a maximum of k times – this strategy will be be denoted by A_k. For now, let's analyze the competitive ratio of A_9.

For request sequence σ, let $|\sigma|$ denote the number of assignments that Alice gets. Then

$$C_{OPT}(\sigma) = \begin{cases} 100|\sigma| & \text{if } |\sigma| < 10 \\ 1000 & \text{if } |\sigma| \geq 10 \end{cases}$$

On the other hand

$$C_{A_9}(\sigma) = \begin{cases} 100|\sigma| & \text{if } |\sigma| < 10 \\ 1900 & \text{if } |\sigma| \geq 10 \end{cases}$$

Hence

$$\frac{C_{A_9}(\sigma)}{C_{OPT}(\sigma)} = \begin{cases} 1 & \text{if } |\sigma| < 10 \\ 1.9 & \text{if } |\sigma| \geq 10 \end{cases}$$

Hence the competitive ratio of A_9 is 1.9.

Next, let's consider whether we can hope to get a better competitive ratio for the RENT OR BUY problem. We'll consider a general setting where the rent cost is an integer R, the buying cost is an integer B. For convenience we assume that R divides B, i.e. B/R is an integer. A deterministic algorithm for the problem can be characterized by when it decides to buy. Note that any competitive algorithm must decide to buy at a finite time, otherwise the competitive ratio would be infinite (why ?).

Suppose the algorithm rents k times before buying. The consider the cost at time $k+1$, i.e just after the algorithm buys for the first time. The cost of the algorithm $C_A(\sigma) = k.R + B$. On the other hand $C_{OPT}(\sigma) = \min\{(k+1)R, B\}$. Now we consider two cases.

Case 1: $B \leq (k+1)R$.

$$\frac{C_A(\sigma)}{C_{OPT}(\sigma)} = \frac{k.R+B}{B} = \frac{(k+1)R - R + B}{B} \geq \frac{2B-R}{B} = 2 - \frac{R}{B}$$

Case 2: $B > (k+1)R$.

$$\frac{C_A(\sigma)}{C_{OPT}(\sigma)} = \frac{k.R+B}{(k+1)R} = \frac{(k+1)R - R + B}{(k+1)R} = 1 + \frac{B-R}{(k+1)R} \geq 1 + \frac{B-R}{B} = 2 - \frac{R}{B}$$

Thus the competitive ratio is at least $2 - \frac{R}{B}$. Substituting $R = 100, B = 1000$, we see that 1.9 is the best competitive we can hope to obtain for Alice's RENT OR BUY problem.

An important thing to note is that we reasoned about deterministic algorithms here. The situation changes if we allow the algorithm to make random choices. Recall that A_k is the algorithm that rents k times before buying. We analyzed A_9 above, and in fact $k = 9$ is the optimal choice for deterministic algorithms. Notice that at time steps $1, \ldots, 9$, the cost incurred by A_9 is equal to the cost of the optimal solution – it is only at time step 10 and beyond that the cost of A_9 jumps up to 1.9 times the optimal cost.

How about algorithm A_k other choices of k ? If we choose $k > 9$, the cost of the algorithm at time $k+1$, right after it buys, is strictly greater than 1.9 times the cost of buying (which is the optimal solution for $|\sigma| > 9$). On the other hand, if we choose $k < 9$, the cost of the algorithm at time $k+1$, right after it buys, is strictly greater than 1.9 times the cost of renting up to that point (which is the optimal solution for $|\sigma| \leq 9$). If we examine A_5 for example, at time step 6 onwards the cost of the algorithm is 1500 – for step 6 and 7, this is more than 1.9 times the optimal cost. However, notice that in the long run (time step 10 and beyond), the cost of A_5 is 1.5 times the optimal cost. Observe that algorithms A_5 and A_9 have their worst performance on disjoint intervals: A_9 performs badly on time steps 10 and beyond, when A_5 does very well, and A_5 does badly on time steps 6 and 7 (possibly also 8 and 9 depending on what competitive ratio we are shooting for) but for these time steps, A_9 performs splendidly. It turns out that we can combine the benefits of the two algorithms and mitigate their weak points by picking randomly amongst them.

Consider algorithm $\tilde{A}$ that runs A_5 with probability 1/2 and runs A_9 with probability 1/2. In other words, it tosses a fair coin initially and decides to run either A_5 or A_9 based on the outcome of the coin toss. Note that the only randomness is in the initial choice of A_5 or A_9 – the algorithm behaves deterministically after that. The expected cost $\mathbb{E}[C_{\tilde{A}}(\sigma)] = \frac{1}{2}C_{A_5}(\sigma) + \frac{1}{2}C_{A_9}(\sigma)$.

$$C_{\tilde{A}}(\sigma) = \begin{cases} 100|\sigma| & \text{if } |\sigma| \leq 5 \\ \frac{1}{2}(1500) + \frac{1}{2}100|\sigma| & \text{if } 5 < |\sigma| < 10 \\ \frac{1}{2}(1500) + \frac{1}{2}(1900) & \text{if } 10 \leq |\sigma| \end{cases}$$

Note that

$$C_{OPT}(\sigma) = \begin{cases} 100|\sigma| & \text{if } |\sigma| \leq 5 \\ 100|\sigma| & \text{if } 5 < |\sigma| < 10 \\ 1000 & \text{if } 10 \leq |\sigma| \end{cases}$$

Hence

$$\frac{C_{\tilde{A}}(\sigma)}{C_{OPT}(\sigma)} = \begin{cases} 1 & \text{if } |\sigma| \leq 5 \\ \frac{750}{100|\sigma|} + \frac{1}{2} \leq \frac{750}{100 \cdot 6} + \frac{1}{2} = 1.75 & \text{if } 5 < |\sigma| < 10 \\ \frac{1700}{1000} = 1.7 & \text{if } 10 \leq |\sigma| \end{cases}$$

Thus the competitive ratio of $\tilde{A}$ is 1.75. It turns out that we can do even better by appropriately combining the algorithms $A_0, \ldots, A_9$. In fact, we have matching upper and lower bounds for the competitive ratio of this problem and we will explore this in precept.

Wrapping up the discussion on Rent or Buy, there are some subtle issues about randomized online algorithms that we swept under the carpet in our discussion above. The competitive ratio of a randomized algorithm is the worst case (over all request sequences σ) of the ratio of expected cost to optimal cost, i.e. $\frac{C_{\tilde{A}}(\sigma)}{C_{OPT}(\sigma)}$. For the purposes of this class, when we analyze the competitive ratio of a randomized algorithm, we assume that the adversary knows the algorithm and can use this information to design a request sequence where the algorithm does badly. However, we assume that the adversary does not know the coin tosses of the algorithm. In other words, the adversary must choose a bad request sequence for the algorithm with knowledge of the "program code" that the algorithm uses but cannot change the request sequence as the execution of the algorithm proceeds.

3 List Update

The field of online algorithms was kicked off by a paper of Danny Sleator and Bob Tarjan[3] in 1985. Their paper was about the analysis of self adjusting data structures, i.e. data structures that adapt to the sequence of operations performed. They introduced the framework of competitive analysis as a method for analyzing the relative performance of different strategies for designing such self adjusting data structures.

To start with, Sleator and Tarjan studied a simple data structure problem called List Update. Here the algorithm is required to maintain a linked list of n items. Requests to the algorithm are of the form Access(x), where x is an item in the list. In response to this request, the algorithm walks the list starting from the first position until the element x is encountered. The algorithm is also allowed to modify the list as it proceeds. In particular, in servicing an Access request, the accessed element may be moved to any earlier position in the list. In addition adjacent elements in the list can be swapped.

In order to analyze this problem under the lens of competitive analysis, we need to be precise about the costs of various operations, so here is a model for the costs:

1. Access(x) where x is currently at position k in the list costs k (this models the cost of traversing the linked list until x is encountered).

2. Moving the accessed element to any earlier position is free.

3. Additional swaps of adjacent elements in the list cost 1. (These are referred to as *paid exchanges*. MTF does not use any such operations).

Here are some natural online algorithms that come to mind:

1. Swap: OnAccess(x), swap x with the element just before it in the list.

2. Move-To-Front (MTF): After Access(x), move x all the way to the front of the list (keeping the relative positions of the other elements unchanged).

3. Frequency Count: Keep track of how many times each element has been accessed so far and maintain the list in decreasing order of these counts (breaking tied arbitrarily).

[3]Yes, that is *our* Bob Tarjan. If you take a data structures and algorithms class, you will enounter many of his fundamental contributions to the field.

It is not very hard to see that SWAP and FREQUENCY COUNT have competitive ratio $\Omega(n)$. Try to construct examples to demonstrate this – we discussed this in class.

We will show that MTF is 2-competitive. It turns out that computing the optimal strategy for LIST UPDATE is a hard problem – in fact it belongs to the class of NP-Complete problems that we will encounter later in the course. It is widely believed that these problems do not have polynomial time algorithms. Given that we don't know how to compute the optimal strategy efficiently, it is quite remarkable that we can compare the cost of MTF to that of the optimal strategy.

How can we prove that MTF is 2-competitive? Let $C_{MTF}(\sigma_k)$ denote the cost of MTF on the kth request and let $C_{OPT}(\sigma_k)$ denote the cost of the optimal algorithm on the kth request. One natural approach would be to prove that for every k, $C_{MTF}(\sigma_k)$ is at most twice $C_{OPT}(\sigma_k)$. If you think about this, you will quickly realize that such a strong statement is not true. The cost of each of the algorithms depends on the position of the currently requested element in their lists, and there is no way we can guarantee that the position in MTF's list is at most twice the position in OPT's list.

The proof of 2-competitiveness is rather ingenious and involves a potential function argument that is quite commonly used in online algorithms. Here is how it works: We imagine running MTF and OPT side by side on the request sequence. The proof defines a certain non-negative function Φ that we call a potential function. Let Φ_k be the value of the potential function after k requests have been serviced. The initial value is $\Phi_0 = 0$. We will show that

$$\underbrace{C_{MTF}(\sigma_k) + \Phi_k - \Phi_{k-1}}_{\text{amortized cost}} \leq 2 \cdot C_{OPT}(\sigma_k) \tag{1}$$

The LHS of the above inequality is referred to as the *amortized cost*. The potential function Φ helps smooth out the costs of the algorithm over various steps so as to facilitate a comparison with the cost of OPT. If the potential function goes up, the amortized cost for that request is higher than the actual cost of MTF. If the potential function goes down, the amortized cost for that request is lower than the actual cost of MTF.

Suppose that the request sequence $\sigma = (\sigma_1, \ldots, \sigma_m)$. Then summing up the above inequality over all $k \in \{1, \ldots, m\}$, we get

$$C_{MTF}(\sigma) + \Phi_m - \Phi_0 \leq 2 \cdot C_{OPT}(\sigma)$$

Here we use $C_{MTF}(\sigma)$ to denote the cost of MTF on the request sequence σ; similarly for $C_{OPT}(\sigma)$. Since $\Phi_0 = 0$ and $\Phi_m \geq 0$, the above inequality implies that

$$C_{MTF}(\sigma) \leq 2 \cdot C_{OPT}(\sigma)$$

So the crux of the matter is establishing the inequality (1) and before that, defining the potential function that facilitates the proof. The potential function we use is defined as follows: Φ is equal to the number of inverted pairs $\{x, y\}$ in the two lists currently maintained by MTF and OPT. $\{x, y\}$ is an inverted pair if x occurs before y in one list, but y appears before x in the other list. Note that $0 \leq \Phi \leq \binom{n}{2}$. We assume that MTF and OPT start out with the exact same ordered list, so the initial value of Φ is zero.

Now we proceed to prove (1).

Proof. Suppose $\sigma_k = x$. We examine the elements that precede x in MTF's list and partition them into two groups depending on where they appear in OPT's list. Define

$$S = \{y|y \text{ precedes } x \text{ in MTF's list and in OPT's list}\}$$

$$T = \{y|y \text{ precedes } x \text{ in MTF's list but not in OPT's list}\}$$

Note that $C_{MTF}(\sigma_k) = |S| + |T| + 1$ since the position of x in MTF's list is $|S| + |T| + 1$.

First, assume that OPT does not make any paid exchanges. $C_{OPT}(\sigma_k) \geq |S| + 1$ since the position of x in OPT's list is at least $|S| + 1$. We claim that $\Phi_k - \Phi_{k-1} \leq |S| - |T|$. Note that MTF moves x to the front of its list. This ensures that inversions involving x and elements of T are destroyed (and this is true irrespective of where OPT moves x) decreasing the potential function by $|T|$. Additionally, new inversions could be created involving x and elements of S. If OPT does not change the position of x, then indeed $|S|$ new inversions are created. If OPT does move x forward, then fewer than S new inversions could be created. In any case, the change in the potential function is bounded by $|S| - |T|$.

Now let's consider the case when OPT makes P paid exchanges in this step. This means that in addition to possibly moving x forward, OPT also swaps P pairs of adjacent elements in its list. Note that each such swap either increases or decreases the potential function by 1, since the swapped pair is the only pair whose orientation changes. In this case, we $C_{OPT}(\sigma_k) \geq |S| + 1 + P$, and $\Phi_k - \Phi_{k-1} \leq |S| - |T| + P$. Now,

$$\begin{aligned} C_{MTF}(\sigma_k) + \Phi_k - \Phi_{k-1} &\leq (|S| + |T| + 1) + (|S| - |T| + P) \\ &= 2|S| + P + 1 \\ &\leq 2(|S| + 1 + P) \leq 2C_{OPT}(\sigma_k) \end{aligned}$$

□

We claim that 2 is the best *deterministic* competitive ratio one can hope to achieve for List Update. More precisely

Theorem 3.1. *Let A be any deterministic algorithm for* List Update*. The for any $\epsilon > 0$, the competitive ratio of A is at least $2 - \epsilon$.*

Proof. (*Sketch*) We construct a request sequence σ by repeatedly requesting the last element in A's list. By design, the cost of A on this request sequence is at least $|\sigma| \cdot n$. We will show that there is an offline algorithm to satisfy the requests with cost at most $|\sigma| \cdot n/2 + \binom{n}{2}$. Thus $C_{OPT}(\sigma) \leq |\sigma| \cdot n/2 + \binom{n}{2}$. By choosing $|\sigma|$ large enough, we can ensure that the ratio of $|\sigma| \cdot n$ to $|\sigma| \cdot n/2 + \binom{n}{2}$ is at least $2 - \epsilon$. (If you work this out, you will see that $|\sigma| \geq 2n/\epsilon$ suffices).

What remains to be shown is that there is an offline algorithm to satisfy the request sequence σ with cost at most $|\sigma| \cdot n/2 + \binom{n}{2}$. In fact this is true for any σ. We simply compute the frequencies of elements in the request sequence and maintain an ordered list in decreasinng order of these frequencies, i.e. most requested elements appearing earlier in the list. We spend at most $\binom{n}{2}$ paid comparisons moving from the initial list configuration to this desired list configuration. Having sorted the list thus, we can show that the average cost for an *Access* operation in σ is at most $n/2$. We leave this proof as an exercise. □

While MTF is an optimal deterministic algorithm for List Update, we can get competitive ratio better than 2 by using randomized algorithms. The following simple algorithm (called BIT) achieves a competitive ratio of 1.75: The algorithm maintains a bit for every element. These bits are initialized randomly to either 0 or 1. When element x is accessed, its bit is flipped (from 0 to 1 and vice versa). If the bit is 1, x is moved to the front, otherwise the list is left unchanged. Interestingly, the algorithm only uses randomness in the beginning. After the bits have been initialized randomly, all further steps are deterministic. To date, we don't have matching upper and lower bounds on the best randomized competitive ratio that can be achieved for List Update.

4 Competitive Ratio revisited

A final note on the definition of competitive ratio: Earlier we said that
Algorithm A is c-competitive if $\boxed{\text{for all request sequences } \sigma,\ C_A(\sigma) \le c \cdot C_{OPT}(\sigma).}$
In fact, this is called *strictly c-competitive.*

An algorithm is said to be c-competitive even if it satisfies a more relaxed condition:
$\boxed{\text{for all request sequences } \sigma,\ C_A(\sigma) \le c \cdot C_{OPT}(\sigma) + \alpha.}$
Here α is a constant – possibly a function of the instance but independent of σ.

Obviously, the notion of strict c-competitiveness is stronger than the relaxed notion of c-competitiveness above. In this course, unless stated otherwise, c-competitive will always mean strictly c-competitive.