Online Algorithms

Lecture Notes for CSCI 570 by David Kempe

December 1, 2006

So far in class, we have only considered the traditional framework for algorithm design, as follows: we are given an input I, are allowed to perform some computation (perhaps restricted in the amount of time), and then produce an output O. For many problems, this is an appropriate framework. However, there are also many cases in the real world in which the algorithm does not know the entire input yet, but still has to make partial decisions about the output.

A natural example is obtained by looking again at one of our applications of weighted interval scheduling. Recall that we were given a list of potential jobs J_i, where each job had a start time of s_i, a finish time of t_i, and a reward of r_i. For instance, these could be job offers of r_i dollars for working from time s_i to t_i. The goal is then to select a set of non-overlapping jobs of maximum total profit. In the real world, we would frequently encounter the situation where the job offers come in one by one, and must be accepted or rejected right away. That is, when the first offer of \$20 for working from 1–3pm arrives, we must decide whether to take it without knowing what (if any) future offers we get. It is easy to see that we could end up making horrendously bad decisions: if we accept the offer, then it could turn out that we get another offer of \$100,000 for the same time interval, which we cannot accept any more. And if we reject the offer, it may be the last offer we ever get. Either way, we could do terribly as a result of lacking clairvoyance. So the problem takes on a distinctly different flavor.

Algorithms which have to make their decisions gradually as data arrives are called *online algorithms*. There are many problems one naturally encounters in this type of setting. Here are a few additional ones, besides many different variations of scheduling and job selection:

1. Response to events or emergencies: Here, we have several "servers" (think of robots, or police cars). Events come in one by one, and we must send a server to each event. Preferably, the server shouldn't have to travel too far to get to the event. If we knew in advance about future events, we would know which of the servers to send, so as to keep others close to where they will be useful next. This problem is called the k-server problem (or also metrical task systems).

2. Data structures (cache, memory, trees, lists): As requests for data come in, we want to keep data that is about to be accessed in places where it can be accessed fast. For instance, the cache should contain frequently (in the future) accessed items. Unfortunately, we do not know which items will be accessed in the future. But the data structure should be updated online anyway.

3. Traffic routing in networks: As requests (s_i, t_i) for connections in a network arrive, we have to decide on a path to route the s_i-t_i traffic. Since we don't know which will be future requests, we may end up congesting edges that are very crucial for other pairs. But in the real world, requests have to be accommodated without knowing future ones.

4. Network design: We sometimes will gradually want to add edges to our network to connect new nodes or new requests, without knowing which other ones will arrive in the future. Thus, we may end up building a suboptimal communication infrastructure, compared to what could have been built knowing all communication requests.

In all these cases (and many others), we are naturally in a scenario where any algorithm will have to make suboptimal decisions at times. Notice that this brings up an interesting contrast with *approximation*

algorithms we studied earlier. In approximation algorithms, we are sub-optimal because we don't have enough resources (usually time, sometimes space) to compute the best solution. In online algorithms, we are sub-optimal because we don't know the future, and thus don't have the entire input available.

Analogous to the *approximation ratio* for approximation algorithms, we define the *competitive ratio* for online algorithms to capture how much worse the algorithm does compared to one that knows about the future.

Definition 1 *Let* OPT *be the optimum offline algorithm (knowing about the future), and* ALG *another algorithm.* c_{OPT} *and* c_{ALG} *denote the costs incurred by those two algorithms.*

1. ALG *is* b-competitive *if there exists a constant* α *such that for all inputs* I*, we have* $c_{\mathrm{ALG}}(I) \leq b \cdot c_{\mathrm{OPT}}(I) + \alpha$*.*
2. ALG *is* strictly b-competitive *if it is* b*-competitive with the constant* $\alpha \leq 0$*.*
3. *The* competitive ratio *of* ALG *is the infimum over all* b *such that* ALG *is* b*-competitive.*

The constant α can be considered a "startup cost" of the algorithm. Frequently, we are more interested in cases where we can get away with no startup cost, which motivates the definition of strictly b-competitive algorithms. The competitive ratio needs to be defined as the infimum (rather than the minimum) because it is conceivable that an algorithm is $2+\epsilon$ competitive for each ϵ, but not 2-competitive. We would still call its competitive ratio 2 in that case.

For online algorithms, we can frequently distinguish two natural types of parameters for inputs: one the size of the underlying structure and the other the length of the "request sequence". For instance, we could think about a constant number k of robots, but a growing number t of requests. Or a constant size (n, m) of a graph, but a growing number of routing requests. Or a constant cache or memory size, but a growing number of requests for individual data items or pages. Whenever this is the case, the constant b for the competitive ratio is allowed to depend on the number that is constant. However, b should not be a function of the number t of requests in the sequence.

1 The Ski Rental Problem

As a first concrete example of an online problem and its solution, we will look at the Ski Rental problem (which abstracts a lot of decisions between renting or buying, and similar investments). In the Ski Rental problem, we assume that we are going skiing for some number d of days total. Each day, we can either rent skis for R dollars, or buy them for $B > R$ dollars. Once we have bought the skis, we can use them for free forever afterwards.

As an offline problem, this is trivial: if $dR < B$, then rent the skis for all d days, otherwise buy them the first day and reuse them. The twist is that we don't know the number d of days in advance. Instead, at the end of each day, it is decided whether we go skiing again. (For instance, we could lose interest, or get seriously injured, or other scenarios may apply.) Thus, the algorithm has to make decisions about whether to buy or rent without knowing how many more days of skiing are coming.

In this case, the algorithm could make some terrible decisions. If we buy on the first day, we might lose interest immediately, and have spent much more money than necessary. The same will apply for most early days. On the other hand, if we keep renting for a long time, it would have been much cheaper to buy right away. A good algorithm should somehow trade off between the two.

First off, we notice that any deterministic algorithm can be described as "rent for d days, then buy", for some d. For as soon as the algorithm buys the skis, it can reuse them for free, so no more decisions need to be made. And there is no additional information beyond the number of days elapsed that the algorithm has available. Executing this algorithm gives us a cost of $B + dR$. We next want to analyze the competitive ratio of this algorithm. We will obtain it by dividing $B + dR$ by the cost of the optimum algorithm. In order to know that cost, we need to know the true number of days of skiing. Here, we can assume without loss of generality that it is $d + 1$. For if it were larger, then the optimum may become more expensive, while the

cost of the algorithm does not change. So the worst case is achieved when the number of days is $d+1$. In that case, the optimum cost is $\min(B,(d+1)R)$, and the ratio is thus

$$\frac{B+dR}{\min(B,(d+1)R)} \quad = \quad 1+\max(\frac{dR}{B},\frac{B-R}{(d+1)R}).$$

The first term in the maximum is monotone increasing in d, and the second monotone decreasing in d. Hence, the best ratio is achieved for the d for which the two terms are equal. For larger d, it would be exceeded by the first term, and for smaller d by the second. Hence, we solve $\frac{dR}{B} = \frac{B-R}{(d+1)R}$. This can be rearranged to $d(d+1) = \frac{B}{R}(\frac{B}{R}-1)$, and hence is solved by $d = \frac{B}{R}-1$. This suggests that the optimum online algorithm is to buy on day number $\frac{B}{R}$. Or, thinking about it otherwise, we buy on the day when renting again would make the total rental cost so far exceed the purchase cost. Substituting the value of d back into the expression for the competitive ratio gives us a ratio of $2-\frac{R}{B}$. So this strategy is strictly 2-competitive. We treated $\frac{B}{R}$ as an integer in defining the online algorithm as "buy on day number $\frac{B}{R}$". If it is not, then the optimum algorithm is either the version rounded up or rounded down.

In the discussion, we assumed that the decision of when we would stop skiing was made in response to our purchase decision. Thus, the world is *adversarial*, and the adversary gets to react to the choices made by the algorithm. At first, this appears to go against the description of having a complete worst-case input, which was shown to the algorithm one by one. However, we should notice that the adversary does not in fact need to react to what the algorithm does. So long as it knows the algorithm, and the algorithm is deterministic, it can simulate it on its own, and decide on the entire worst-case input ahead of time, being assured that the actual execution will match the simulation. Notice that this makes the adversary quite powerful. However, this approach is motivated by our desire to design algorithms that will work well even if everything goes wrong. It does not imply a philosophical judgment about whether the world is indeed adversarial, but rather a design principle of guarding against even the worst possible coincidences.

Notice also that this illustrates the power of randomization. If the algorithm generates random bits for its decisions (for instance, choosing randomly when to buy the skis), then the adversary cannot be assured that any two executions will behave the same, and cannot design as easily an input on which the algorithm will always incur high cost. For instance, we showed above that no deterministic online algorithm can achieve a competitive ratio better than $2-\frac{R}{B}$. It is not too difficult to show that the expected cost of a randomized ski rental algorithm can be better than this bound. Of course, if the adversary actually gets to observe the coin flips, then it may be able to adapt its input accordingly. But for the most part, randomization is a good way of improving the performance of the algorithm. We will not cover randomized algorithms in this class, but an example of a randomized online algorithm is discussed in Chapter 13.8 of the textbook [3].

Finally, we should mention that while the competitive ratio is a natural measure of the performance of an algorithm, it is not the only one. For instance, one could argue that once we reach day d, having rented every day so far, the rental cost is really a sunk cost, and we should treat the problem as though this were again the first day. This would lead to the strategy of always renting. Obviously, this does not achieve a constant competitive ratio, but it does appear to be cheapest on a day-by-day basis. We will not concern ourselves here with measures other than the competitive ratio, but it should be noticed that they are frequently equally well motivated.

2 List Accessing

Among the first papers to study online algorithms was one by Sleator and Tarjan [5], studying algorithms for online list accessing and paging. More generally, this falls into the area of self-organizing data structures, which try to keep frequently accessed items cheaply accessible. A more complex such data structure are Splay Trees.

Here, we focus on accessing the elements of a linked list. Specifically, if the i^{th} element of the list is accessed, then the cost incurred for this access is i. Thus, frequently accessed elements should be toward the front of the list. The problem, of course, is that we do not know which items will be requested in the future. In the absence of this knowledge, one could think of several natural online heuristics:

1. The *static* heuristic does not rearrange the list at all. If we were to choose a static ordering, we would likely want to sort the list by access frequencies. In fact, if all accesses were chosen *randomly* according to a known frequency, then pre-sorting the list by decreasing frequencies would be an optimal algorithm.

2. The *move-to-front (*MTF*)* heuristic always moves the most recently accessed element to the front of the list.

3. The *frequency-count (*FC*)* heuristic keeps a count of the number of times f_i each element i has been accessed, and keeps the list sorted by decreasing f_i.

4. The *transposition (*TRANS*)* heuristic always moves the most recently accessed element one position forward, by swapping it with its neighbor.

5. One could come up with several other variants between MTF and TRANS, such as moving an element halfway to the front, swapping it two positions forward, or other similar ideas.

In the early 1980s, these heuristics were analyzed mostly through experiments, or with probabilistic analysis. These assumed certain distributions on the requests for elements (which were not known to the online heuristic in advance), and compared the heuristics under those distributions. For instance, Rivest [4] showed that if accesses are independently random with a given probability distribution p, then the total expected cost of TRANS is at most that of MTF. However, experimental results (e.g., [1]) showed that MTF performed significantly better than TRANS, and was at least competitive with FC.

Motivated by this discrepancy, Sleator and Tarjan proposed analyzing heuristics in terms of their competitive ratio. They posited the following model: The request sequence is $\sigma(1), \sigma(2), \sigma(3), \ldots$. The sequence is of length t, but the value of t, as well as any values $\sigma(j)$ for $j > i$, are not known to the algorithm in round i. If $\sigma(i)$ is in position k, then the access cost is k. Afterwards, the element can be moved forward (closer to the front) for free. Additional rearrangements can be made at a cost of 1 per transposition of two adjacent elements. No additional swaps can be made that involve non-adjacent elements.

While this model may seem somewhat arbitrary at first, it models fairly naturally the fact that once $\sigma(i)$ has been found in position k, the elements 1–k have already been searched, so inserting $\sigma(i)$ anywhere in between will not incur much additional cost. (It turns out that charging a constant for this move does not alter the result.) Other elements, on the other hand, need to be found first to be swapped, and thus, such transpositions are more expensive. Before we proceed to analyze MTF in this model, let us briefly look at the competitive ratio of TRANS and FC.

1. For TRANS, we could have a sequence that always accesses the element currently last in the list. This results in repeated swaps between the last two elements. The total cost of t such accesses is tn. On the other hand, the optimum solution could move the two elements to the first two positions after (or before) their first accesses, and incur a total cost of at most $2n + 2t$. The $2n$ term is moving two elements by n positions each, and the $2t$ term is accessing two elements a total of t times, each at cost at most 2. Thus, the competitive ratio is at least $\frac{tn}{2(n+t)}$, which converges to $n/2$ as $t \to \infty$. This is clearly a very bad competitive ratio, as any algorithm will achieve ratio n.

2. For FC, the problem is that once frequencies have been counted, it adapts very slowly to new patterns. For instance, assume that we choose $k \gg n$. The sequence accesses the first element k times, then the second element $k-1$ times, and so forth, and finally the n^{th} element $k+1-n$ times. As a result, the FC heuristic will never reorganize the list. Its total cost is thus

$$k + 2(k-1) + 3(k-2) + \ldots + n(k+1-n) \quad \geq \quad (k-n) \cdot \tfrac{n(n+1)}{2} \quad = \quad \Omega(kn^2).$$

   On the other hand, the optimum solution would move each element to the front when it is accessed for the first time, and thus incurs cost at most $n^2 + nk = O(nk)$. The first term is moving n elements forward at most n each, and the second term is accessing each of n elements at most k times each, at cost 1. Thus, the competitive ratio is at least $\Omega(kn^2)/O(nk) = \Omega(n)$, i.e., also linear in the list size n.

In light of these negative results, it is perhaps a bit surprising that the simple MTF heuristic has much better competitive ratio.

Theorem 2 MTF *is strictly 2-competitive. That is, for all request sequences σ, we have $c_{\mathrm{MTF}}(\sigma) \leq 2c_{\mathrm{OPT}}(\sigma)$.*

Proof. We define $c_{\mathrm{MTF}}(i)$ and $c_{\mathrm{OPT}}(i)$ as the costs incurred by MTF and OPT in the i^{th} step. Thus, $c_{\mathrm{MTF}}(\sigma) = \sum_{i=1}^{t} c_{\mathrm{MTF}}(i)$, and $c_{\mathrm{OPT}}(\sigma) = \sum_{i=1}^{t} c_{\mathrm{OPT}}(i)$. Ideally, we would like to show the inequality term by term, i.e., show that $c_{\mathrm{MTF}}(i) \leq 2c_{\mathrm{OPT}}(i)$ for all i. But it is easy to see that this will not work: the optimum solution could after the first step rearrange its list at great cost. Then, subsequent steps for MTF will be significantly more expensive. Since a step-by-step comparison does not work, we need to take into account the effect that an expensive current round has for future accesses. If an expensive access right now means that future accesses will be cheaper, then we should apply a "discount" for comparison purposes. This is exactly the idea of *amortized analysis*, which we already saw earlier in the class when looking at Fibonacci Heaps.

To define this "discount" more formally, we observe that if the lists of OPT and MTF are identical, then accesses to any element cost exactly the same, so not only will the factor 2 bound hold, but there will be cost to spare. Thus, good moves by MTF are ones that make the lists more similar. Notice that those are exactly the ones that are very expensive compared to OPT: if OPT had a cheap access, then the element was close to the front. Thus, moving the element to the front will make the lists much more similar in this case. We will measure similarity of the lists with a non-negative *potential function* $\phi(i)$. It will capture numerically how similar the two lists are at the end of the i^{th} access. When the lists are identical, $\phi(i) = 0$ will hold. In order to grant a bonus to moves making the lists more similar, we define the *amortized cost* $c'_{\mathrm{MTF}}(i) := c_{\mathrm{MTF}}(i) + \phi(i) - \phi(i-1)$. This makes the amortized cost cheaper whenever the lists became more similar (and more expensive when they became more dissimilar).

By rearranging the definition, we obtain that $c_{\mathrm{MTF}}(i) = c'_{\mathrm{MTF}}(i) + \phi(i-1) - \phi(i)$, and thus the total cost of MTF is

$$\sum_{i=1}^{t} c_{\mathrm{MTF}}(i) \quad = \quad \sum_{i=1}^{t} (c'_{\mathrm{MTF}}(i) + \phi(i-1) - \phi(i)) \quad = \quad \phi(0) - \phi(t) + \sum_{i=1}^{t} c'_{\mathrm{MTF}}(i) \quad \leq \quad \sum_{i=1}^{t} c'_{\mathrm{MTF}}(i).$$

The second equality holds because the series telescopes (all terms except the first $\phi(i-1)$ and the last $-\phi(i)$ cancel out). The inequality follows because $\phi(0) = 0$, and $\phi(t) \geq 0$. Thus, it will be sufficient to show that the amortized cost satisfies $c'_{\mathrm{MTF}}(i) \leq 2c_{\mathrm{OPT}}(i)$, and indeed, that is what we will show.

Of course, we first need to define a potential function, since so far, all the discussion has been in the abstract. Finding the right potential function for analyzing an online algorithm is frequently difficult, and involves a lot of experimentation. Here, we want a function that measures how similar two orderings of a list are. Several such measures exist. Letting $p_{\mathrm{MTF}}(j)$ and $p_{\mathrm{OPT}}(j)$ denote the positions in the list at which element j is located, two natural candidates are the following:

1. Spearman's Footrule: $\sum_j |p_{\mathrm{OPT}}(j) - p_{\mathrm{MTF}}(j)|$ is the sum of distances between the positions in the two lists, for all elements.

2. Kendall's tau: $|\{(i,j) \mid p_{\mathrm{OPT}}(i) < p_{\mathrm{OPT}}(j) \text{ and } p_{\mathrm{MTF}}(j) < p_{\mathrm{MTF}}(i)\}|$ is the total number inversions, i.e., the number of pairs of elements which are in one relative order in MTF's list, and in the other order in OPT's list.

Both are natural measures (as are several others), and there is no a priori reason to prefer one over the other. The only way to figure out which is the right potential function is to try and prove the algorithm correct, and either succeed or fail. In this case, it turns out that Spearman's Footrule does not work, while Kendall's tau does. Hence, from now on, we let $\phi(i)$ denote the Kendall tau distance between the two lists after the i^{th} access is over.

We now want to analyze the i^{th} step, where element $\sigma(i)$ is accessed. Assume that it is located in position m in the MTF list, and in position k in the OPT list. Thus, the respective access costs are m and k. In

addition, MTF moves $\sigma(i)$ to the front, thus eliminating some inversions and creating others. Also, OPT may move $\sigma(i)$ forward, and perform some paid transpositions.

First, let us analyze the move that MTF makes. By moving $\sigma(i)$ to the front, MTF eliminates all inversions with elements which are behind position k in OPT's list, but before position m in MTF's list. Suppose that there are $v \geq 0$ of them. Then, the remaining $m-1-v$ elements preceding $\sigma(i)$ in MTF's list must also precede $\sigma(i)$ in OPT's list. In particular, this proves that $k \geq m - v$.

While the move to the front eliminates v inversions, it also may create some new ones. All of the $m-1-v$ elements that were previously before $\sigma(i)$ in both of the lists are now still before $\sigma(i)$ in OPT's list, but behind $\sigma(i)$ in MTF's list. Thus, the move has created $m-1-v$ new inversions. Thus, the net change in inversions is $m-1-2v$, and the total amortized cost incurred by MTF, summing both the actual cost and the change in inversions, is $m + (m-1-2v) = 2(m-v) - 1$.

In addition, suppose that OPT makes f free moves forward, and p paid adjacent transpositions. Each move forward actually eliminates one inversion for us. Each paid transposition can introduce at most one new inversion. Thus, the total amortized cost is $2(m-v) - f - 1 + p \leq 2(m-v+p) \leq 2(k+p)$. The last inequality holds because we proved above that $k \geq m - v$. And $2(k+p)$ is exactly twice the cost incurred by the optimum solution, so we proved that the inequality holds for each i. ■

One interesting question not discussed in this context is what the optimum offline solution (knowing the future) actually looks like. It turns out that it is still open whether it can be computed in polynomial time. Neither NP-hardness nor an optimal polynomial-time algorithm are known. Of course, we also just proved that MTF is a 2-approximation algorithm.

Another interesting variation is to look at different cost structures. For instance, we could posit that accessing the first $m < n$ elements costs 1, while accessing the remaining elements costs $C \gg 1$. This would naturally model a two-tiered memory structure such as cache vs. main memory, or main memory vs. external memory. A lot of work has been done in this area on competitive paging strategies. Here, we can only scratch the surface, but interested students may want to have a look at the textbook by Borodin and El-Yaniv [2].

References

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

[2] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis.* Cambridge University Press, 1998.

[3] J. Kleinberg and E. Tardos. *Algorithms Design.* Addison-Wesley, 2005.

[4] R. Rivest. On self-organizing sequential search heuristics. *Communications of the ACM*, 19:63–67, 1976.

[5] D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.