On-line Algorithms

Introduction

An online algorithm is one that can process its input piece-by-piece in a serial fashion, i.e., in the order that the input is fed to the algorithm, without having the entire input available from the start.
In contrast, an offline algorithm is given the whole problem data from the beginning and is required to output an answer which solves the problem at hand. (For example selection sort requires that the entire list be given before it can sort it, while insertion sort doesn't.)

Online algorithms make a sequence of decisions under uncertainty. One of the most powerful methods of analyzing such problems is by competitive analysis.

Definition: α-competitive: A has a competitive ratio of α if there exist a constant b such that for every sequence of request (σ): $A(\sigma) \leq \alpha.\ OPT(\sigma) + b$

Definition: Strictly α-competitive: A has a strictly competitive ratio of α if for every sequence of request (σ): $A(\sigma) \leq \alpha.\ OPT(\sigma)$

Rent or Buy Problem

Assume we decide to go skiing for the first time. Buying the equipment costs about \$500 and renting it for a weekend costs \$50. Should we buy or rent?

Clearly, it depends on how many more times we are going to go skiing in the future. At any time, the only input for the algorithm is the fact that this is the t-th time we are going skiing, and that we have been renting so far; the algorithm decides whether to buy or rent based on t.

What are the competitive ratios of the possible choices of t?
$t=1 \rightarrow$ The competitive ratio is 10, because we always spend \$500, and if it so happens that we never go skiing again after the first time, then the optimum is \$50.

$t=2 \rightarrow$ The competitive ratio is 5.5, because if we go skiing twice then we rent the first time and buy the second (in the worst case scenario), spending a total of \$550, but the optimum is \$100.

In general, for every $t < 10$, the competitive ratio is

$$\frac{500+50(t-1)}{50t} = 1 + \frac{9}{t}$$

If $t \geq 10$, then the competitive ratio is

$$\frac{500+50(t-1)}{500} = 0.9 + \frac{t}{10}$$

So the best choice of t is $t = 10$, which gives the competitive ratio 1.9.

The Secretary Problem

Suppose we have joined an online dating site, and that there are n people that we are rather interested in. We would like to end up dating the best one. We could go out with all of them but a decision about each person is to be made immediately after meeting and if we are dating someone, we are not going to go out with anybody else.
How can we maximize the probability of ending up dating the best person?

Algorithm: We are going to pick a random order of the n people, and go out, one after the other, with the first n/e people. In these first n/e dates, we make no decision. The purpose of this first phase is to calibrate our expectations. After these n/e dates, we continue to go out on more dates following the random order, but as soon as we found someone who is better than everybody we have seen so far, that's the one we are going to pick. It can be shown that this strategy picks the best person with probability about 1/e, which is about 37%.

There is another online algorithm for this problem that picks a person of average rank O(1), and which is then competitive for the optimization problem of minimizing the rank. The algorithm is rather complicated, and it is based on first computing a series of timesteps $t_0 \leq t_1 \leq \ldots$ according to a rather complicated formula, and then proceeds as follows: we reject the first t_0 people, then if we find someone in the first t_1 dates which is better than all the previous people, we pick that person. Otherwise, between the t_1+1-th and the t_2-th date, we are willing to pick someone if that person is either the best or the second best of those seen so far, and so on. Basically, as time goes on, we reduce our expectations accordingly.

Paging and Caching

Consider a two-level memory system that consists of a small fast memory and a large slow memory. Here, each request species a page in the memory system. A request is served if the corresponding page is in fast memory. If a requested page is not in fast memory, a page fault occurs. Then a page must be moved from fast memory to slow memory so that the requested page can be loaded into the vacated location.
A paging algorithms species which page to evict on a fault. If the algorithm is online, then the decision which page to evict must be made without knowledge of any future requests. The cost to be minimized is the total number of page faults incurred on the request sequence.

Greedy Paging (offline): Given an input sequence $\rho_0, \rho_1 \ldots \rho_n$; On a miss, replace the page whose next occurrence is the farthest in the sequence.

Lemma: Greedy paging is optimal and $f_{OPT}([\rho_0, \rho_1, \ldots, \rho_n]) = n/k$.

There are three well-known deterministic online paging algorithms:

- LRU (Least Recently Used): On a fault, evict the page in fast memory that was requested least recently.
- FIFO (First-In First-Out): Evict the page that has been in fast memory longest.
- LFU (Least Frequently Used): Evict the page that has been requested least frequently.

Theorem: Suppose that, for a certain sequence of requests, the optimal sequence of choices for size-h cache causes m misses. Then, for the same sequence of requests, LRU for a size-k cache causes at most $\frac{k}{k-h+1}.m$ misses.

Theorem: The algorithms LRU and FIFO are k-competitive.

Randomized online paging algorithm:

Algorithm Marking: At the beginning of each phase, all pages in the cache are unmarked. Whenever a page is requested, it is marked. On a miss, a page is chosen uniformly at random from among the unmarked pages in the cache, and this page is evicted. A phase ends when all the pages in fast memory are marked and a miss occurs. Then, all marks are erased and a new phase is started.

Theorem: The marking algorithm is $2H_k$-competitive where H_k is the k-th Harmonic number.

List Update

There is a list of n elements in the form of a linked list. The cost of accessing the k-th element is k. We also have flexibility of rearranging the list as we access elements. More specifically, we can choose to perform the following actions when accessing the k-th element:

- Move the k-th element to any position in front of it for free.
- Swap consecutive elements in the list for cost of 1.

The task we want to solve is given a sequence signal; design an online algorithm to minimize cost of accessing elements in this signal.

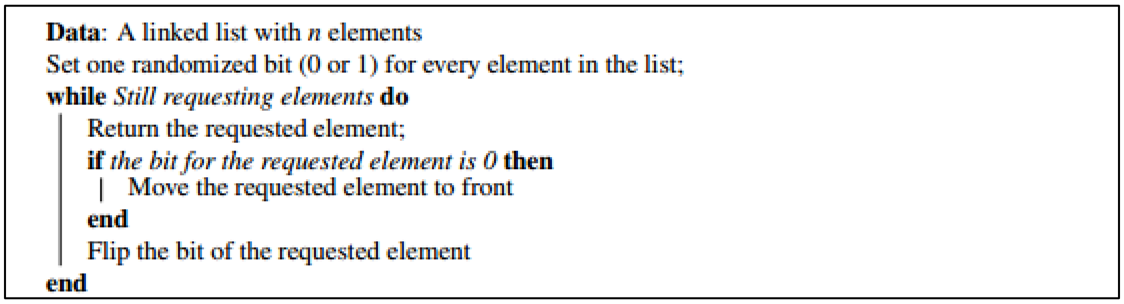
Algorithms:
Here are some examples of possible deterministic algorithms to solve list update:

- MTF (Move To Front): After accessing the k-th element, move it all the way to the very front of the list.
- MoTF (Move one To Front): After accessing the k-th element, move it one position forward in the list.
- Frequently Count: Keep the list sorted with respect to frequency.

Theorem: MTF is 2-competitve.
Theorem: The lowest competitive ratio for any deterministic algorithm to solve list update is 2.

Randomized online algorithm for list update:

The best known randomized algorithm to solve List Hash has a competitive ratio of approximately 1.6. Here is one randomized algorithm for this problem (Algorithm BIT):



```
Data: A linked list with $n$ elements
Set one randomized bit (0 or 1) for every element in the list;
while Still requesting elements do
    Return the requested element;
    if the bit for the requested element is 0 then
        Move the requested element to front
    end
    Flip the bit of the requested element
end
```

The only randomization BIT has is the randomized bits assigned to each element in the very beginning. BIT is 1.75-competitive.

References:

[1] http://www.cs.princeton.edu/courses/archive/spring13/cos521/notes/lecture5.pdf
[2] http://www2.informatik.hu-berlin.de/~albers/papers/brics.pdf
[3] http://lucatrevisan.wordpress.com/2011/03/09/cs261-lecture-17-online-algorithms/