Online Algorithm

Xiaofeng Gao

Department of Computer Science and Engineering
Shanghai Jiao Tong University, P.R.China

Algorithm Course @ Shanghai Jiao Tong University

Outline

Online Algorithm

An online problem is one where not all the input is known at the beginning.

Rather, the input is presented in stages, and the algorithm needs to process this input as it is received.

As the algorithm does not know the rest of the input, it may not be able to make optimum decisions.

Example 1: Renting vs. Buying Skis

Example: A simple example of an online problem is the ski-rental.

A skier can either rent skis for a day or buy them once and for all. Buying a pair of skis costs k times as much as renting it. Also, skis once bought cannot be returned.

Whenever the skier goes skiing, she does not know whether she will ski again or how many times. This makes the problem an online one.

Example 1:Renting vs. Buying Skis (Cont.)

Consider the following scenario: A malicious Weather God lets the skier ski as long as she is renting skis, but does not let her go on any skiing trips after she has bought them.

Suppose the skier decides to rent skis on the first k' trips and buys them on the next trip. The total cost incurred by the skier is $k' + k$. But the optimum cost is k if $k' + 1 \geq k$ and $k' + 1$ otherwise.

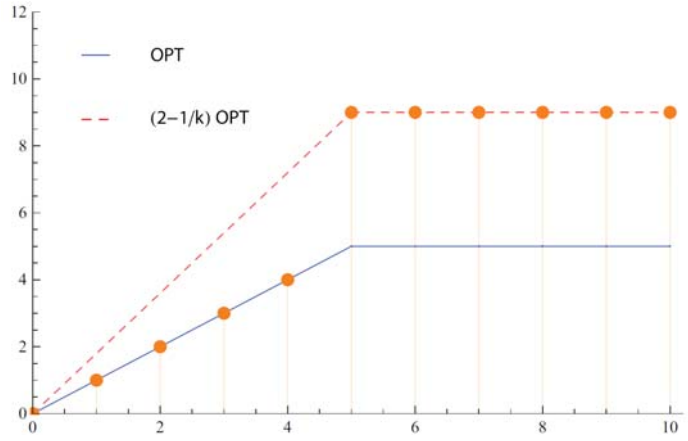
Therefore, the skier ends up paying a factor of $(k' + k)/\min\{k, k' + 1\}$ as much as the optimum.

Choosing $k' = k - 1$, this factor equals $2 - 1/k$.

This factor $2 - 1/k$ is called the Competitive Ratio of the algorithm. Note that while calculating this factor, we used the worst possible input sequence.

Example 1:Renting vs. Buying Skis (Cont.)

The figure is a plot for $k = 5$. Our algorithm starts with points on OPT, but must eventually contain a point on the dotted line. This tells us the minimum multiple of OPT we can achieve.



Competitive Ratio

Definition: an online algorithm A is α-competitive if for all input sequences σ,

$$C_A(\sigma) \leq \alpha \cdot C_{OPT}(\sigma) + \delta$$

where C_A is A's cost function, C_{OPT} is the optimum's cost function, and δ is some constant. We can look upon the competitive ratio as the inherent cost of not knowing the future.

Example 2: Paging

Example: Paging is a more realistic example of an online problem.

We have some number of pages, and a cache of k pages. Each time a program requests a page, we need to ensure that it is in the cache.

We incur zero cost if the page is already in the cache. Otherwise, we need to fetch the page from a secondary store and put it in the cache, kicking out one of the pages already in the cache. This operation costs one unit.

The online decision we need to make is which page to kick out when a page fault occurs. The initial contents of the cache are the same for all algorithms.

Example 2: Paging

We have learned that FF (Furthest in the Future) is optimal offline eviction algorithm in Lecture-Greedy Algorithm.

Some online strategies are:

- LIFO (Last In First Out)
- FIFO (First In First Out)
- LRU (Least Recently Used — the page that has not been requested for the longest time is kicked out)
- LFU (Least Frequently Used)

LRU is k-competitive for the Online Paging Problem

Proof: Define a phase as a sequence of requests during which LRU faults exactly k times. All we need to show is that during each phase, the optimal algorithm (OPT) must fault at least once.

Case 1: LRU faults at least twice on some page σ_i during the phase. In this case, σ_i must have been brought in once and then kicked out again during the phase.

When σ_i is brought in, it becomes the most recently used page. When it is later kicked out, σ_i must be the least recently used page.

Therefore, each of the other $k-1$ pages in the cache at this time must have been requested at least once since σ_i was brought in.

Counting σ_i and the request that causes σ_i to be evicted, at least $k+1$ distinct pages have been requested in this phase.

Therefore, OPT must fault at least once.

LRU is k-competitive for the Online Paging Problem

Case 2: LRU faults on k different pages.

If this is the first phase, then OPT must fault at least once, since OPT and LRU start off with the same pages in their cache.

Else, let σ_i be the last page requested in the previous phase. This page has to be in both LRU's and OPT's cache at the start of the phase.

If LRU does not fault on σ_i, then the k pages on which LRU faults cannot all be present in OPT's cache at the beginning of the phase.

Therefore, OPT must fault at least once.

The case when LRU does fault on σ_i is similar to Case 1. □

Upper Bound using FF

Let A be an algorithm for the paging problem.

Consider a malicious program that uses only $k+1$ pages. At each step, the program requests the one page that is not in A's cache.

So the cost incurred by A is $|\sigma|$ (the number of the requests).

FF (Furthest in the Future) is a strategy that kicks out the page that is going to be requested furthest in the future.

When FF kicks out a page σ_i, the next page fault can occur only when this page is requested again, since there are only $k+1$ pages.

Upper Bound using FF

But by the definition of FF, each of the $k-1$ pages that was not kicked out at the first page fault must be requested before σ_i is requested again.

Thus, page faults are at least a distance k away and the cost of FF is at most $(|\sigma|-1)/k+1$.

Therefore A can be no more than k-competitive.

LRU is k-competitive. So LRU is optimal online paging algorithm

k is not a good competitive ratio at all, since cache sizes can typically be very large.

However, LRU performs very well in practice.

The Online Packing-Covering Framework

For simplicity, we consider a simpler setting in which $b(j)=1$ and $a(i,j)\in\{0,1\}, i\in S(j)$ if $a(i,j)=1$

Primal (Covering) problem:

$$\begin{aligned}&\text{minimize:} && \textstyle\sum_{i=1}^{n} c_i x_i\\ &\text{subject to:}\\ &\forall 1\le j\le m: && \textstyle\sum_{i\in S(j)} x_i \ge 1\\ &\forall 1\le i\le n: && x_i\ge 0\end{aligned}$$

Dual (Packing) problem:

$$\begin{aligned}&\text{maximize:} && \textstyle\sum_{j=1}^{m} y_j\\ &\text{subject to:}\\ &\forall 1\le i\le n: && \textstyle\sum_{j:i\in S(j)} y_j \le c_i\\ &\forall 1\le j\le m: && y_j\ge 0\end{aligned}$$

The Online Packing-Covering Framework

Primal (Covering) problem:

$$\begin{aligned}&\text{minimize:} && \textstyle\sum_{i=1}^{n} c_i x_i\\ &\text{subject to:}\\ &\forall 1\le j\le m: && \textstyle\sum_{i=1}^{n} a(i,j) x_i \ge 1\\ &\forall 1\le i\le n: && x_i\ge 0\end{aligned}$$

Dual (Packing) problem:

$$\begin{aligned}&\text{maximize:} && \textstyle\sum_{j=1}^{m} y_j\\ &\text{subject to:}\\ &\forall 1\le i\le n: && \textstyle\sum_{j=1}^{m} a(i,j) y_j \le c(i)\\ &\forall 1\le j\le m: && y_j\ge 0\end{aligned}$$

The Online Packing-Covering Framework

Primal (Covering) problem:

Known: The cost function.

Given one-by-one: The linear constraints.

Difference: It may not decrease any previously increased variable.

Dual (Packing) problem:

Known: The values $c_i (1\le i\le n)$.

Unknown: The profit function, the exact packing constraints and new variable y_j .

Difference: The algorithm may increase the value of a variable y_j only in the round in which it is given.

Basic Discrete Algorithm

Algorithm 1: Basic discrete algorithm

1 Upon arrival of a new primal constraint $\sum_{i \in S(j)} x_i \geq 1$ and the corresponding dual variable y_j;
2 **while** $\sum_{i \in S(j)} x_i < 1$ **do**
3 For each $i \in S(j) : x_i \leftarrow x_i \left(1 + \frac{1}{c_i}\right) + \frac{1}{|S(j)|c_i}$;
4 $y_j \leftarrow y_j + 1$;

The algorithm produces:

- A (fractional) covering solution which is feasible and $O(\log d)$-competitive.
- An (integral) packing solution which is 2-competitive and violates each packing constraint by at most a factor of $O(\log d)$.

Basic Continuous Algorithm

Algorithm 2: Basic continuous algorithm

1 Upon arrival of a new primal constraint $\sum_{i \in S(j)} x_i \geq 1$ and the corresponding dual variable y_j;
2 **while** $\sum_{i \in S(j)} x_i < 1$ **do**
3 Increase the variable y_j continuously;
4 For each variable x_i that appears in the (yet unsatisfied)primal constraint increase x_i according to the following function: $x_i \leftarrow \frac{1}{d}\left[\exp\left(\frac{\ln(1+d)}{c_i} \sum_{j:i \in S(j)} y_j\right) - 1\right]$;

The algorithm produces:

- A (fractional) covering solution which is feasible and $O(\log d)$-competitive.
- A (fractional) packing solution which is feasible and $O(\log d)$-competitive.

Continuous Algorithm

Algorithm 3: Continuous Algorithm

1 Upon arrival of a new primal constraint $\sum_{i \in S(j)} x_i \geq 1$ and the corresponding dual variable y_j;
2 **while** $\sum_{i \in S(j)} x_i < 1$ **do**
3 Increase the variable y_j continuously;
4 If $x_i = 0$ and $\sum_{j | i \in S(j)} y_j = c_i$ then set $x_i \leftarrow \frac{1}{d}$;
5 For each variable x_i, $\frac{1}{d} \leq x_i < 1$, that appears in the (yet unsatisfied)primal constraint increase x_i according to the following function: $x_i \leftarrow \frac{1}{d} \exp\left(\frac{\sum_{j:i \in S(j)} y_j}{c_i} - 1\right)$;

The algorithm produces:

- A (fractional) $O(\log d)$-competitive covering solution.
- A (fractional) 2-competitive packing solution and violates each packing constraint by a factor of at most $O(\log d)$.