Online Algorithms

Motivations

Sometimes one has to react before everything is known. For example, in stock trading one has a limited amount of total funds, and has to make buy/sell decisions without knowing the future stock prices. This requires the study of online algorithms.

Online Algorithms

A more formal definition follows. The input of an online algorithm is a sequence of requests $\sigma = \sigma(1), \dots, \sigma(n)$. The algorithm needs to serve these requests in order. When serving $\sigma(t)$, the algorithm does not know the future requests $\sigma(t')$ for $t' > t$.

In contrast, an offline algorithm can make decisions after knowing the complete sequence of requests. Clearly, an offline algorithm has huge advantage comparing to online algorithm. (Imagine what will happen if you knows the future stock prices.)

Competitive Ratio

The goal of the online algorithm is usually to minimize a cost function. The cost of an online algorithm's output is often compares with the optimal cost of the best possible offline algorithm. For a sequence of request σ, let $OPT(\sigma)$ and $ALG(\sigma)$ denote the cost incurred by the best possible offline algorithm OPT, and the online algorithm ALG, respectively. The **competitive ratio** of an online algorithm ALG is defined as

$$\max_{\sigma} \frac{ALG(\sigma)}{OPT(\sigma)}$$

Because of the maximum in the definition, this is a worst case analysis.

Secretary Problem

This is a widely used example. The problem is also known as the marriage problem, the sultan's dowry problem, the fussy suitor problem, the googol game, and the best choice problem.

Let's use a more neutral imaginary situation to explain this problem. You are allowed to take only one from a bag of n diamonds, with different sizes that you do not know. Each time a diamond is randomly taken from the bag and shown to you. You now need to decide to take it or reject it. If you take it the game stops; otherwise, the diamond is put away and a new one is taken from the bag. You cannot go back to take a diamond that has been put away.

What's your best strategy to maximize the probability of getting the largest diamond? Assume that you only care about getting the best – getting the second best has no weight in your evaluation.

The optimal strategy is fairly simple: reject the first $\frac{n}{e}$ diamonds. After that take the first diamond that is larger than all diamonds you've ever seen.

We won't prove that this is the best strategy; but only show that this strategy is the best among the same class of strategies: observing k candidates first and then taking the first one better than all of them.

Theorem: Among the class of observing k candidates first and later taking the first one better than observed, the optimal value of k is $\frac{n}{e}$, and the algorithm gets the largest diamond with probability $\frac{1}{e} \approx 36.8\%$.

Proof: For a uniformly random permutation of the diamonds, let i be the index of the largest diamond. When $i \leq k$ the algorithm will fail. But when $i > k$, the algorithm succeeds if and only if "the second largest diamonds among the first i falls in the first k". Since this is a uniformly random permutation,

$$\Pr(\text{success}|i \text{ is the largest}) = \Pr(\text{second largest of first } i \text{ is in the first } k \mid i \text{ is the largest}) = \frac{k}{i-1}$$

So

$$\Pr(\text{success}) = \sum_{i=k+1}^{n} \Pr(\text{success}|i \text{ is the largest}) \cdot \Pr(i \text{ is the largest}) = \sum_{i=k+1}^{n} \frac{k}{i-1} \cdot \frac{1}{n}$$
$$= \frac{k}{n} \cdot \sum_{i=k+1}^{n} \frac{1}{i-1} = \frac{k}{n} \cdot \sum_{i=k}^{n-1} \frac{1}{i} \approx \frac{k}{n} \cdot \ln\left(\frac{n-1}{k-1}\right)$$

Let $x = \frac{k}{n}$, elementary calculus can show that $x \cdot \ln\frac{1}{x}$ takes the maximum value $\frac{1}{e}$ when $x = \frac{1}{e}$.

QED.

For small n, the optimal k can be computed by dynamic programming. The following table shows the optimal k and its success probability for small n:

n	1	2	3	4	5	6	7	8	9
k	0	0	1	1	2	2	2	3	3
prob	1	.5	.5	.458	.433	.428	.414	.410	.406

Paging problem

Computer systems often have multi-levels of storage. Earlier days, there were RAM and hard drives (or tapes). Now, there were L1-cache, L2-cache, and RAM. The idea is to cache the frequently used data from slow but large storage to fast but small storage. This potentially speeds up the overall access speed.

For the sake of algorithm study, let's assume we are working on a two-tier system with a cache (RAM) and disk (slow). The paging problem models the data with pages. Each access to the page

has two possibilities: in the cache or not. If in cache, then the access is served. If not, then a "page fault" happens. One old page in the cache is removed to create space, and the desired page on the disk is loaded to the cache. Since page fault is expensive we want to minimize its occurrences.

Suppose the cache has space for k pages, given a sequence of page accesses: $p_1, p_2, \dots, p_n$, the algorithm needs to decide which page to remove from the cache whenever a page fault happens. The goal is to minimize the total number of page faults.

Two scenarios arise: online and offline. The offline scenario assumes the sequence $p_1, p_2, \dots, p_n$ is known before the algorithm makes the plan. The online scenario assume the algorithm has to respond to a page fault immediately when it occurs at time t, without the knowledge of p_i for $i > t$. Note that this may create a worst case scenario where a removed page at time t is accessed immediately at time $t + 1$.

MIN algorithm

MIN algorithm is an **offline** algorithm. Let $p_1, \dots, p_n$ be the access sequence. When a page fault occurs at time t, assume the cache has the set of pages S_t, then the algorithm remove the page in S_t that is accessed the latest (or never accessed) after t.

The algorithm was first introduced in (L. A. Belady. A study of replacement algorithms for virtual storage computers. IBM Systems Journal, 5(2):78–101, 1966.). The proof of the optimality first appeared in (R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. IBM Systems Journal, 9(2):78–117, 1970.) The proof was quite complicated. A simpler proof can be found at http://web.stanford.edu/~bvr/pubs/paging.pdf. Here we examine another proof (with the same idea I believe).

Theorem 1. The MIN algorithm produces the minimum number of page faults.

Proof: Let $p_1, \dots, p_n$ be the access sequence. We prove by induction on the sequence length n. It is clear that the theorem is true for $n = 1$.

For any algorithm ALG, let S_t be the set of pages in the cache at time t before the page access p_t. Thus, $S_1, S_2, \dots, S_n$ is the sequence of cache states if ALG is used. We will prove that this sequence is no better than the one provided by MIN.

If ALG makes the same removal decision at time 1, then our proof is reduced to a length $n - 1$ access sequence that starts at time 2. By induction, the theorem is true. Thus we focus on the case where ALG violates the MIN strategy at time 1.

Also, when page fault occurs at time 1, if there is a page $p \in S_1$ that is never accessed afterwards, then ALG can be modified to adopt the MIN strategy at time 1 to make the result better. The theorem is also proved by induction.

Thus, we assume at time 1, there is a $p_j \in S_1$ that is the latest accessed. Let j is its first occurrence in the access sequence. Assume that ALG chooses to remove another element that occur before j. Suppose it is p_i for $1 < i < j$.

We also assume that p_j is first removed by ALG at time i'. $i' = \infty$ if it's never removed.

Case 1. $i' \geq i$

I modify $S_1, S_2, \dots, S_n$ to a different sequence:

$$S_1, S_2 - p_j + p_i, \dots, S_i - p_j + p_i, S_{i+1} - p_j, \dots, S_j - p_j, S_{j+1}, \dots, S_n$$

That is, I replace p_j with p_i up to time i. This is still a valid cache state sequence. At time i, the new sequence will save a page fault. After time i, I follow ALG, except that now p_j is not in cache. This may create at most one additional page fault at time j. After that, everything is the same. So, I successfully modified ALG. It does not make the result worse, but now it follows MIN strategy at time 1. By induction, the theorem is true.

Case 2. $i' < i$.

The only problem this condition creates is that at time $i' < i$, after removing p_j, $S_{i'+1} - p_j + p_i$ is larger than the cache size. So what I do instead is to remove p_i at time i'. That is, I modify $S_1, S_2, \dots, S_n$ to a different sequence:

$$S_1, S_2 - p_j + p_i, \dots, S_{i'} - p_j + p_i, S_{i'+1}, \dots, S_n$$

Note that this will be again a valid state sequence. The number of page faults remains the same. But now, the new sequence follows the MIN strategy at time 1. By induction, the theorem is true.

QED.

LRU (Least Recently Used)

Although the MIN is offline optimal, it is usually not practical. We need online algorithms. There are a few commonly used strategies:

1. FIFO: First in first out. Remove the earliest page in the cache. This strategy is used by Windows NT, 2000.
2. LRU: Least Recently Used. Remove the page in the cache that is the least recently used. This is somewhat like our brain's memory. We forget things we studied in college that we haven't used for a long time.
3. Randomized: Choose a page at random (according to some strategy) to remove.

FIFO and LRU are deterministic algorithms. Their performance is measured by the competitive ratio (worst case analysis, over all possible access sequences). Both have the competitive ratio k, where k is the size of the cache. FIFO is easier to implement than LRU. The latter has to update

the time stamp and recalculate the oldest page for each access (think of a data structure for this); whereas FIFO is a simple queue. But they may have different "average" behavior depending on the distribution of the access sequence.

Next let us examine LRU's competitive ratio in further detail. To generalize, we compare the performance of LRU and MIN on different cache sizes, n_{LRU} and n_{MIN}, respectively. Reason for this will become clear later. We limit the discussion for $n_{MIN} \leq n_{LRU}$. (Otherwise?)

Theorem 2: For any access sequence, $F_{LRU} \leq \left(\frac{n_{LRU}}{n_{LRU}-n_{MIN}+1}\right) F_{MIN} + n_{MIN}$.

Proof: After the first access of page p, LRU and MIN both have p in their cache, and p is the most recently accessed page. Now consider a subsequent subsequence t such that LRU faults $f \leq n_{LRU}$ times.

In this access sequence, if any page p' faults at least twice, then during these two faults, there are accesses to at least n_{LRU} different pages other than p'. So there are at least $n_{LRU} + 1$ different pages. The same is true if page p faults at least once. To access $n_{LRU} + 1$ pages, MIN needs to fault at least $n_{LRU} - n_{MIN} + 1$ times.

If none of the above two cases occurs, then the f faults are on different pages that are not p. MIN has at most $n_{MIN} - 1$ of them in cache initially. So MIN must fault at least $f - n_{MIN} + 1$ times.

Now divide the access sequence into phases. Each phase contains exactly n_{LRU} faults for LRU, except for the first phase. From the above discussion, the ratio between F_{LRU} and F_{MIN} is at most $\frac{n_{LRU}}{n_{LRU}-n_{MIN}+1}$ for each phase except phase 1. In phase 1, if LRU faults f times, then MIN faults at least $f - n_{MIN}$ times. Thus the theorem is proved.

Q.E.D.

Now let's come back to the relation between n_{LRU} and n_{MIN}. When $n_{LRU} = n_{MIN}$, that is, the cache sizes of the two algorithms are the same. The competitive ratio of LRU is only n_{LRU}, the cache size. This is a pathetic bound. (Imagine you have millions of pages in cache.)

However, if you think the same question in a different way, say, when $n_{LRU} = 2 \times n_{MIN}$, things are not bad. Because now the ratio becomes 2. That is, LRU is relatively good comparing to the optimal algorithm with a much smaller cache size.

The bound is actually tight, as indicated in the following theorem.

Theorem 3. Let A be any online algorithm with n_A pages in cache. There is a arbitrarily long access sequence s such that $F_A(s) \geq \left(\frac{n_A}{n_A-n_{MIN}+1}\right) F_{MIN}(s)$.

Proof: Let's first construct a sequence of length n_A so that A has n_A faults and MIN has $n_A - n_{MIN} + 1$ faults. To make it arbitrarily long, we only need to repeat the sequence.

The first $n_A - n_{MIN} + 1$ accesses are to distinct pages that are neither in A nor in MIN's caches. So, they all fault. These $n_A - n_{MIN} + 1$ pages plus the original n_{MIN} pages give a set S of $n_A + 1$ pages, larger than A's cache. So, for each of the next $n_{MIN} - 1$ access, we (the adversary) can always choose a page from S to produce a fault for A. So, $F_A(s) = n_A$. However, since MIN is an offline algorithm, it avoids any faults in the next $n_{MIN} - 1$ accesses by not removing them from the cache. So, $F_{MIN}(s) = n_A - n_{MIN} + 1$.

Q.E.D.

Theorems 2 and 3 indicate that LRU achieves the best possible competitive ratio. In fact, the FIFO algorithm also achieves the same competitive ratio. This is given as an assignment.

Randomization

Although the competitive ratio is pathetic for paging problem, it works well in practice. This can be thought as due to the distribution of the access sequence. In another word, the average case performance may not be as bad as the worst case analysis. But average analysis makes (sometimes arbitrary) assumption on the input's distribution. So our fate is at the mercy of the adversary. Imagine that a hacker (adversary) might send in a hard problem to exhaust the resource of your system.

One way to defeat the adversary is randomization: we don't rely on the input's random distribution. Rather, we make clever random choices in our decision process. So the adversary won't be able to design a deterministic sequence to easily defeat the algorithm.

Of course, this requires the adversary to produce the sequence in advance – after knowing our algorithm, but before the algorithm responds to the sequence. This is called an **oblivious adversary**.

MARKING algorithm: The algorithm requires an auxiliary bit (mark) for each page in cache. Initially all pages in cache are unmarked. When a request to page p arrives, if it is in cache then mark it. If not then remove an unmarked page at random, fetch p from disk, and mark it. If all pages in cache are marked when a page faults, unmark all pages first before applying the above strategy.

Theorem 4. MARKING is $2H_k$-competitive. Here k is the size of cache, and $H_k = 1 + \frac{1}{2} + \cdots + \frac{1}{k}$ is the k-th harmonic number.

Proof: Fix a sequence of n pages requests $p_1, \ldots, p_n$. Starting from time 1, decompose the sequence into phases sequentially. Each phase (except for the last one) $p_i, \ldots, p_{j-1}$ is such that

1. there are exactly k distinct pages in the phase.
2. adding p_j will increase it to $k + 1$ distinct pages.

This way, the MARKING algorithm starts each phase with no page marked (or unmark all pages first), and ends the phase with all page marked.

Consider phase i. A page p accessed in the phase is **old** if it was already accessed in phase $i-1$. Otherwise it is **new**.

Because of marking, each distinct page induces at most 1 fault in a phase. Each new page induces precisely one page fault. Each old page p induces 0 or 1 page fault. It induces 0 fault if and only if it is still in the cache when it is first accessed in current phase.

Let m_i denote the number of new pages in phase i; and $k-m_i$ be the number of old pages. The worst case scenario is that all the new pages are accessed before any old page. The expected number of page faults under this scenario provides an upper bound.

After all m_i new pages are accessed, let $o_1, \ldots, o_{k-m_i}$ be the old pages in the order of their first access in current phase. When accessing o_j the first time, there are still $k-m_i-j+1$ old pages that are unmarked. This is a random subset of the $k-j+1$ old pages (original cache minus $o_1, \ldots, o_{j-1}$). So, o_j falls in this set with probability $\frac{k-m_i-j+1}{k-j+1}$. Therefore, it induces a fault with probability

$$1-\frac{k-m_i-j+1}{k-j+1}=\frac{m_i}{k-j+1}.$$

So the expected number of page faults is

$$m_i+\sum_{j=1}^{k-m}\frac{m_i}{k-j+1}=m_i\times\left(1+\sum_{j=m+1}^{k}\frac{1}{j}\right)\leq m_i\times H_k$$

On the other hand, in the current and the previous phase, in total $k+m_i$ distinct pages are requested. So an optimal algorithm must pay at least m_i page faults. Summing this for all adjacent pairs of phases, and taking the double counting into account, an optimal algorithm must pay at least $\frac{1}{2}\times\sum_i m_i$ faults.

Q.E.D.

Theorem 5. No randomized algorithm can be better than H_k-competitive.

Proof omitted for now.

So the MARKING algorithm is close to the optimal. In fact, there is another algorithm called PARTITION that achieved the competitive ratio lower bound H_k. We do not study this algorithm here.

Three Types of Adversaries

Besides the oblivious adversary, there are two other scenarios:

Oblivious adversary: the adversary generates the request sequence in advance, and serves the request sequence offline.

Since a randomized algorithm is not totally predictable, randomization makes it hard for the oblivious adversary to generate a difficult request sequence for the algorithm.

Adaptive online adversary: the adversary makes the next request after knowing the previous responses of the algorithm. It also serves the request immediately. The performance of the algorithm is compared with the adversary's service.

For deterministic online algorithm, this adversary is the same as the oblivious adversary, because the algorithm's behaviour is predictable.

Adaptive offline adversary: the adversary makes the next request after knowing the previous responses of the algorithm. It serves the sequence offline.

This adversary is so strong. Intuitively, randomization tries to hide the algorithm's response from the adversary. But when making the next request, an adaptive adversary knows the algorithm's previous responses (except that the future responses are uncertain). In fact, under this adversary, randomization adds no power to the online algorithm.

There is a famous result from (Ben-David et. al, On the Power of Randomization in Online Algorithms, STOC 1990) saying that randomization has almost no power against the adaptive online and offline adversaries. We give the two main theorems but omit the proofs.

Theorem 6. If there is a randomized algorithm that is α-competitive against any adaptive offline adversary then there also exists an α-competitive deterministic algorithm.

Theorem 7. If G is a c-competitive randomized algorithm against any adaptive online adversary, and there is a randomized d-competitive algorithm against any oblivious adversary, then G is a randomized $(c \cdot d)$-competitive algorithm against any adaptive offline adversary.

The two theorems basically indicate that randomization provides no power against the adaptive offline adversary, and provides little (if any) power against the adaptive online adversary.

Combining Experts Opinions

Recall that we used stock trading as an example for the motivation for the model of online algorithms. For stock trading, there are many "experts" who make predictions about the market. Some of them are better than others during a certain period of time. We can learn this with hindsight. But we can't go back to the old time to follow this best expert's trading opinion. So, can we have an online algorithm?

More formally, there are n experts: $E_1, \dots, E_n$. At the beginning of each time $t = 1, \dots, m$, each expert gives an opinion on something (e.g., 0 or 1). The online algorithm needs to decide which

opinion (0 or 1) to follow. Once decision made, the truth is revealed and a constant cost incurred if the opinion is wrong. The online algorithm wants to minimize the total cost.

Is there a competitive online algorithm?

Clearly, if the "experts" are allowed to make mistakes, then the online algorithm can't be competitive – an offline algorithm that knows all the truth will make 0 mistakes. So, let's ask a more meaningful question:

Is there an online algorithm that is competitive against the best expert with hindsight? More specifically, given a sequence, let $c(E_i)$ be the number of errors made by expert E_i. Let $OPT = \min_{i=1..n} c(E_i)$. Can we bound the errors of the online algorithm by a function of OPT?

There is such an algorithm called weighted majority.

Weighted Majority: Each expert E_i has a weight w_i that is initially 1. The algorithm makes decision by a weighted majority of the experts' opinions:

$$argmax_{d=0,1} \sum_{i\ s.t.\ E_i\ says\ d} w_i$$

Whenever a truth is known, the weight of each expert who made a mistake is halved.

See the following example:

					prediction	correct
weights	1	1	1	1		
predictions	Y	Y	Y	N	Y	Y
weights	1	1	1	.5		
predictions	Y	N	N	Y	N	Y
weights	1	.5	.5	.5		

Theorem 6. The number of errors made by Weighted Majority is at most $2.41 \times (OPT + \log_2 n)$.

Proof: Suppose E_{opt} is the best expert. Let W be the total weight of all experts at a certain time. Whenever the algorithm makes a mistake, the wrong experts contribute at least $\frac{W}{2}$ in the total weight. Thus W is reduced by at least $\frac{W}{4}$. Suppose the algorithm makes ALG mistakes after n rounds, then the total weight is no more than $\left(\frac{3}{4}\right)^{ALG} \times n$.

Meanwhile, the best expert's final weight would be $\left(\frac{1}{2}\right)^{OPT}$. So,

$$\left(\frac{1}{2}\right)^{OPT} \leq \left(\frac{3}{4}\right)^{ALG} \times n$$

By taking a logarithm we have the theorem.

QED.

So, this isn't bad at all. If indeed there is an expert who makes very few mistakes, then the online algorithm will start to perform well (up to a constant factor) after about $O(\log n)$ steps.

Can we do better?

One way is to learn more slowly. In another word, do not penalize an expert too much after a mistake. Say, we reduce the weight w to $(1-\epsilon)w$ after a mistake. It is easy to verify the result of Th6 becomes

$$ALG \leq OPT \times \log_{\left(\frac{2-\epsilon}{2}\right)}(1-\epsilon) + \log_{\left(\frac{2}{2-\epsilon}\right)} n$$

By letting $\epsilon \to 0$, the coefficient $\log_{\left(\frac{2-\epsilon}{2}\right)}(1-\epsilon)$ approaches 2. But at the same time, $\log_{\left(\frac{2}{2-\epsilon}\right)} n$ is getting much larger. Thus, the "warm-up" time is longer, but the overall performance on a very long sequence is a bit better.

Can we do still better? For example, let the coefficient approach 1.

The answer is randomization.

Randomized Weighted Majority: The only modification to the weighted majority algorithm is the following: At each time, suppose the weights of the experts are $w_1, \ldots, w_n$ and $W = \sum_{i=1}^{n} w_i$ be the total weight. Then the algorithm follows expert E_i's opinion with probability $\frac{w_i}{W}$.

First a technical preparation: recall the Taylor series:

$$f(x) = f(0) + \frac{f'(0)}{1!} \times x + \frac{f''(0)}{2!} \times x^2 + \cdots$$

So,

$$\log(1+x) = 0 + x - \frac{1}{2} \cdot x^2 + O(x^3)$$

Theorem 7. The expected number of mistakes made by the randomized weighted majority algorithm is upper bounded by $(1+\epsilon) \times OPT + \frac{1}{\epsilon} \times \log n$.

Proof: Suppose at time $t = 1, \ldots, m$, a fraction of q_t weights make mistakes. Then the expected number of mistakes made by the algorithm $ALG = \sum_{t=1}^{m} q_t$. We want to upper bound ALG.

Let W_t be the total weight after time t. Moreover, $W_0 = n$ be the initial total weight. During time t, the $q_t \cdot W_{t-1}$ weights with mistakes are reduced to $(1-\epsilon) \cdot q_t \cdot W_{t-1}$. In another word, a portion of $\epsilon \cdot q_t$ of the total weight is removed. Thus,

$$W_t = W_{t-1} \times (1 - \epsilon \cdot q_t)$$

So, the final total weight

$$W_m = W_0 \times \prod_{t=1}^{m}(1 - \epsilon \cdot q_t) = n \times \prod_{t=1}^{m}(1 - \epsilon \cdot q_t)$$

Take logarithm, we have

$$\log W_m = \log n + \sum_{t=1}^{m} \log(1 - \epsilon \cdot q_t) \leq \log n + \sum_{t=1}^{m} -\epsilon \cdot q_t = \log n - \epsilon \times ALG$$

On the other hand, the final weight of the optimal expert is $(1 - \epsilon)^{OPT}$. Thus,

$$(1 - \epsilon)^{OPT} \leq W_m$$

Take a log at both side, we get

$$OPT \times \log(1 - \epsilon) \leq \log W_m \leq \log n - \epsilon \times ALG$$

So,

$$ALG \leq -\frac{\log(1 - \epsilon)}{\epsilon} \times OPT + \frac{1}{\epsilon} \times \log n \approx \left(1 + \frac{\epsilon}{2}\right) \times OPT + \frac{1}{\epsilon} \times \log n$$

QED.

When $\epsilon = \frac{1}{2}$, $ALG \leq 1.39 \times OPT + 2 \log n$.

When $\epsilon = \frac{1}{4}$, $ALG \leq 1.15 \times OPT + 4 \log n$.

When $\epsilon = \frac{1}{8}$, $ALG \leq 1.07 \times OPT + 8 \log n$.

If set $\epsilon = \left(\frac{\log n}{OPT}\right)^{\frac{1}{2}}$, the two terms in the theorem balance. And the expected mistakes are bounded by $OPT + 2 \cdot (OPT \times \log n)^{\frac{1}{2}}$. There is only a small o additive error. If not knowing OPT, since $OPT \leq m$, we can set $\epsilon = \left(\frac{\log n}{m}\right)^{\frac{1}{2}}$ to get $OPT + 2 \cdot (m \times \log n)^{\frac{1}{2}}$.

The above algorithm can be extended so that it works even if the opinions are not binary; or if the opinions' losses are a function of the opinions that have real value in $[0,1]$. We will do this in the assignment.