Introduction to Algorithms

Online Algorithm

Haisheng Tan

School of Computer Science and Technology
University of Science and Technology of China (USTC)

Fall Semester 2023

Introduction

Online Algorithms

Online Algorithms are algorithms that need to make decisions without full knowledge of the input, i.e., with full knowledge of the past but no (or partial) knowledge of the future.

For this type of problem we will attempt to design algorithms that are *competitive* with the optimal offline algorithm, which has perfect knowledge of the future.

Competitive Ratio

- Competitive ratio: For the maximization problems,

$$ratio = \max_{s} \frac{ALG(s)}{Offline\ OPT(s)}$$

, where $ALG(s)$ is the cost of ALG on the input sequence s and Offline $OPT(s)$ is the optimal cost for the same sequence with full information.

- Competitive ratio is a worst case bound.

The Ski-Rental Problem

- Assume that you are taking ski lessons. After each lesson you decide (depending on how much you enjoy it, what is your bones status, and the weather) whether to continue to ski or to stop totally.
- You have the choice of either renting skis for 1$ a time or buying skis for B$.
- Will you buy or rent?

The Ski-Rental Problem

- If you knew in advance how many times T you would ski in your life then the choice of whether to rent or buy is simple. If you will ski more than B times then buy before you start, otherwise always rent.
- The cost of this algorithm is min(T, B).
- This type of strategy, with perfect knowledge of the future, is known as an offline strategy.

The Ski-Rental Problem

- In practice, you don't know how many times you will ski. What should you do?
- An online strategy will be a number k such that after renting $k-1$ times you will buy skis (just before your k^{th} visit).

Claim:

Setting $k = B$ guarantees that you never pay more than twice the cost of the offline strategy.

Example: Assume $B = 7\$$ Thus, after 6 rents, you buy. Your total payment: $6 + 7 = 13\$$

The Ski-Rental Problem

Claim:

Setting $k = B$ guarantees that you never pay more than twice the cost of the offline strategy.

Proof:

When you buy skis in your k^{th} visit, even if you quit right after this time, $T \geq B$.

- Your total payment is $\mathrm{k-1+B=2B-1}$.
- The offline cost is $\mathrm{min(T,B)=B}$.
- The ratio is $(2B-1)/B = 2-1/B$.

We say that this strategy is (2-1/B)-competitive.

The Ski-Rental Problem

Is there a better choice of k?

- Let k be any strategy (buy after **k-1** rents).
- Suppose you buy the skis at the k^{th} time and then break your leg and never ski again.
- Your total ski cost is $k-1+B$ and the optimum offline cost is **min(k,B)**.
- For every k, the ratio $(k-1+B)/min(k,B)$ is at least $(2-1/B)$.
- Therefore, every strategy is at least $(2-1/B)$- -competitive.

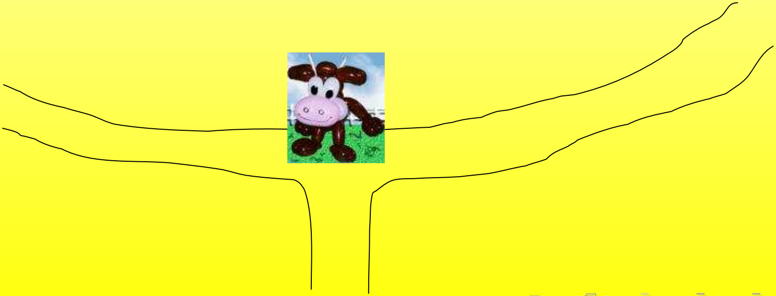
The Ski-Rental Problem

General Rule 1:

When balancing small incremental costs against a big one-time cost, you want to delay spending the big cost until you have accumulated roughly the same amount in small costs.

The Lost Cow Problem

Old McDonald lost his favorite cow. It was last seen marching towards a junction leading to two infinite roads. None of the witnesses can say if the cow picked the left or the right route.



The Lost Cow Problem

OLD MCDONALDS ALGORITHM()

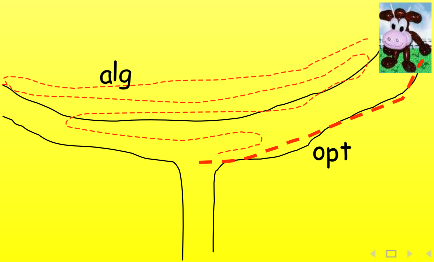
```
d = 1;current side = right
while true do
    Walk distance d on current side
    if find cow then
        exit
    else
        d = 2d
        Flip current side
        return to starting point
```

The Lost Cow Problem

Theorem

Old McDonald'algorithm is 9-competitive.

In other words: The distance that Old McDonald might pass before finding the cow is at most 9 times the distance of an optimal offline algorithm (who knows where the cow is.).



The Lost Cow Problem

Theorem

Old McDonald'algorithm is 9-competitive.

Proof:

The worst case is that he finds the cow a little bit beyond the distance he last searched on this side (why?) [1].

Thus, OPT $2^j + \varepsilon$ where $j = \#$ of iterations and ε is some small distance. Then,

$$Cost\ OPT = 2^j + \varepsilon > 2^j$$

$$\begin{aligned} Cost\ ON &= 2(1+2+4+...+2^{j+1}) + 2^j + \varepsilon \\ &= 2 \cdot 2^{j+2} + 2^j + \varepsilon = 9 \cdot 2^j + \varepsilon < 9 \cdot Cost\ OPT \end{aligned}$$

[1] *Note: this implies that at the first try, you search the direction where the cow is.*

The Secretary Problem

We have n candidates (perhaps applicants for a job or possible marriage partners). Our goal is choose the very best candidate. The assumptions are

- Candidates can be totally ordered from best to worst with no ties.
- Candidates arrive sequentially in random order.
- We can only determine the relative ranks of the candidates as they arrive. We cannot observe the absolute ranks.
- After each interview we must either immediately accept or reject the applicant. Once a candidate is rejected, she can not be recalled. Once a candidate is accepted, we stopped interviewing.
- The number of candidates n is known.

The Secretary Problem

An Online Strategy

- After meeting the i-th candidate, we are able to give a score denoted $\text{score}(i)$.
- Selecting a positive integer $k < n$, interviewing and then rejecting the first k candidates.
- Accept the first candidate thereafter who has a higher score than all k preceding candidates.
- If it turns out that the best-qualified candidate was among the first k interviewed, then we have to accept the n-th applicant.

The Secretary Problem

ON-LINE-MAXIMUM(k, n)

```
1: bestscore = −inf
2: for i = 0 to k do
3:     if score(i) > bestscore then
4:         bestscore = score(i)
5: for i = k+1 to n do
6:     if score(i) > bestscore then return i
   return n
```

The Best Possible k

Let S be the event that we succeed in choosing the best-qualified candidate. We choose the k to maximize $Pr\{S\}$.
Let S_i be the event that we succeed when the best-qualified applicant is the i-th one interviewed. We assume k is fixed.

$$Pr\{S\} = \sum_{i=1}^{n} Pr\{S_i\} = \sum_{i=k+1}^{n} Pr\{S_i\}$$

Let B_i be the event that the best-qualified applicant must be in position i, and let O_i be the event that none of the applicants in positions from $k+1$ to $i-1$ are chosen. Then,

$$Pr\{S_i\} = Pr\{B_i \cap O_i\} = Pr\{B_i\}Pr\{O_i\}$$

The best possible k

The maximum score is equally likely to be in any one of the n positions.

$$Pr\{B_i\} = 1/n$$

For event O_i to occur, the maximum value in positions from 1 to $i-1$ must be in one of the first k positions.

$$Pr\{O_i\} = k/(i-1)$$

$$Pr\{S_i\} = Pr\{B_i\}Pr\{O_i\} = k/(n(i-1))$$

And we can calculate the possibility of S.

$$Pr\{S\} = \sum_{i=k+1}^{n} Pr\{S_i\} = \sum_{i=k+1}^{n} \frac{k}{n(i-1)} = \frac{k}{n}\sum_{i=k+1}^{n} \frac{1}{i-1} = \frac{k}{n}\sum_{i=k}^{n-1} \frac{1}{i}$$

The best possible k

We have

$$\int_{k}^{n} \frac{1}{x}\,dx \leq \sum_{i=k}^{n-1} \frac{1}{i} \leq \int_{k-1}^{n-1} \frac{1}{x}\,dx$$

Evaluating these definite integrals.

$$\frac{k}{n}(\ln n - \ln k) \leq Pr\{S\} \leq \frac{k}{n}(\ln(n-1) - \ln(k-1))$$

Choosing k that maximizes the lower bound on $Pr\{S\}$.

$$\frac{1}{n}(\ln n - \ln k - 1)) = 0 \rightarrow k = \frac{n}{e}$$

If we implement our strategy with $k = n/e$, we succeed in hiring our best-qualified applicant with probability at least $1/e$.

The Lost Cow Problem & The Secretary Problem

General Rule 2:

When lack of key information, make some tentative actions, or accumlate partial information first.

Reading List

Lecture:

Advanced Algorithms, CMU. `http://www.cs.cmu.edu/~15850/`.

Book:

Online Computation and Competitive Analysis, Allan Borodin & Denis Pankratov, Cambridge University Press.

Online Scheduling and Load Balancing

Problem Statement

- A set of m identical machines,
- A sequence of jobs with processing times $p_1, p_2, \cdots$
- Each job must be assigned to one of the machines.
- When job j is scheduled, we dont know how many additional jobs we are going to have and what are their processing times.

Goal

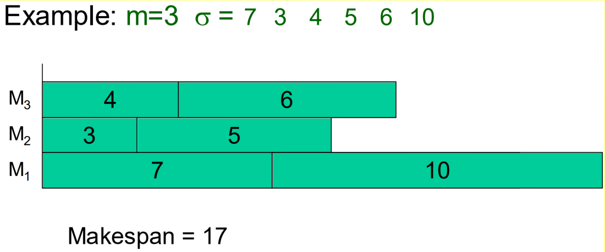
Schedule the jobs on machines in a way that minimizes the makespan $= max_i \cdot \sum_{j \text{ on } M_i} p_j \cdot$ (the maximal load on one machine).

Online Scheduling and Load Balancing

List Scheduling

A greedy algorithm: always schedule a job on the least loaded machine.

Example:



Online Scheduling and Load Balancing

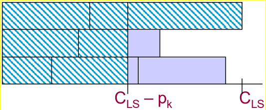
Theorem

List- Scheduling is $(2 - 1/m)$ - competitive.

Proof:

Let H_j denote the last completion time on the j^{th} machine. Let k be the job that finishes last and determines C_{LS}. All the machines are busy when j starts its processing, thus, $\forall j, H_j \geq C_{LS} - p_k$. For at least one machine (that processes k) $H_j = C_{LS}$.

$$\sum_i p_i = \sum_j H_j \geq (m-1)(C_{LS} - p_k) + C_{LS}$$
$$\sum_i p_i + (m-1)p_k \geq mC_{LS}.$$
$$C_{LS} \leq 1/m \sum_i p_i + p_k(m-1)/m.$$



Online Scheduling and Load Balancing

$$C_{LS} \leq 1/m \sum_i p_i + p_k(m-1)/m.$$

Consider an optimal offline schedule.
$\mathrm{C_{opt}} \geq \mathrm{max_i\, p_i} \geq \mathrm{p_k}$ (some machine must process the longest job).
$C_{\mathrm{opt}} \geq 1/m \sum_i p_i$ (if the load is perfectly balanced).
Therefore,

$$C_{LS} \leq C_{opt} + C_{opt}(m-1)/(m) = (2-1/m)C_{opt}.$$

Online Scheduling

Are there any better algorithms?
Not significantly. Randomization do help.

	deterministic			randomized	
m	lower bound	upper bound	LS	lower bound	upper bound
2	1.5	1.5	1.5	1.334	1.334
3	1.666	1.667	1.667	1.42	1.55
4	1.731	1.733	1.75	1.46	1.66
∞	1.852	1.923	2	1.58	...

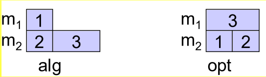
A lower Bound for Online Scheduling

Theorem

For $m = 2$, no algorithm has $r < 1.5$.

Proof:

Consider the sequence $\sigma = 1, 1, 2$. If the first two jobs are scheduled on different machines, the third job completes at time 3.



$C_A = 3, \quad C_{opt} = 2$

$r = 3/2$

If the first two jobs are scheduled on the same machine, the adversary stops.

$C_A = 2, \quad C_{opt} = 1$

$r = 2$

Paging-Cache Replacement Policies

Problem Statement

- There are two levels of memory:
  - fast memory M_1 consisting of k pages (cache)
  - slow memory M_2 consisting of n pages $(k < n)$.
- Pages in M_1 are a strict subset of the pages in M_2.
- Pages are accessible only through M_1.
- Accessing a page contained in M_1 has cost 0.
- When accessing a page not in M_1, it must first be brought in from M_2 at a cost of 1 before it can be accessed. This event is called a page fault.

Paging-Cache Replacement Policies

Problem Statement (cont.)

If M_1 is full when a page fault occurs, some page in M_1 must be evicted in order to make room in M_1.

How to choose a page to evict each time a page fault occurs in a way that minimizes the total number of page faults over time?

Paging-An Optimal **Offline** Algorithm

Algorithm LFD (Longest-Forward-Distance)

An optimal off-line page replacement strategy.
On each page fault, replace the page in M_1 that will be requested farthest out in the future.

Example:

Example: M_2={a,b,c,d,e} n=5, k=3

σ= a, b, c , d , a , b , e , d , e , b , c , c , a , d

a	a	a	a	e	e	e	e	c	c	c	c
b	b	b	b	b	b	b	b	b	b	a	a
c	d	d	d	d	d	d	d	d	d	d	d
	*			*					*		*

4 cache misses in LFD

Paging-An Optimal **Offline** Algorithm

A classic result from 1966

LFD is an optimal page replacement policy.

Proof idea:

For any other algorithm A, the cost of A is not increased if in the 1^{st} time that A differs from LFD we evict in A the page that is requested farthest in the future.

However, LFD is not practical.

It is not an online algorithm!

Online Paging Algorithms

FIFO: first in first out: evict the page that was entered first to the cache.

Example: M_2={a,b,c,d,e} n=5, k=3

σ= a, b, c , d , a , b , e , d , e , b , c , c , a , d

a	d	d	d	e	e	e	e	e	e	a	a
b	b	a	a	a	d	d	d	d	d	d	d
c	c	c	b	b	b	b	b	c	c	c	c
	*	*	*	*	*			*		*	

7 cache misses in FIFO

Theorem

FIFO is $k-competitive$: for any sequence, #misses(FIFO) $\leq k$ #misses (LFD)

Online Paging Algorithms

LIFO: last in first out: evict the page that was entered last to the cache.

Example: M_2={a,b,c,d,e} n=5, k=3

σ= a, b, c , d , a , b , e , d , e , b , c , c , a , d

a	a	a	a	a	a	a	a	a	a	a	a
b	b	b	b	b	b	b	b	b	b	b	b
c	d	d	d	e	d	e	e	c	c	c	d
	*			*	*	*		*			*

6 cache misses in LIFO

Theorem

For all $n > k$, LIFO is not competitive: For any c, there exists a sequence of requests such that #misses(LIFO) $\geq c$ #misses (LFD)

Online Paging Algorithms

LRU: least recently used: evict the page with the earliest last reference.

Example: M_2={a,b,c,d,e} n=5, k=3

σ= a, b, c , d , a , b , d , e , d , e , b , c

a	d	d	d	d	d	d	d	d	c
b	b	a	a	a	e	e	e	e	e
c	c	c	b	b	b	b	b	b	b
	*	*	*		*				*

Theorem

LRU is $k-competitive$.

Paging-a bound for any deterministic online algorithm

Theorem

For any k and any deterministic online algorithm A, the competitive ratio of $A \geq k$.

Proof:

Assume $n = k + 1$ (there are $k + 1$ distinct pages).
What will the adversary do?
Always request the page that is not currently in M_1
This causes a page fault in every access. The total cost of A is $|\sigma|$.

Paging-a bound for any deterministic online algorithm

What is the price of LFD in this sequence?

- At most a single page fault in any k accesses (LFD evicts the page that will be needed in the $k+1^{\text{th}}$ request or later)
- The total cost of LFD is at most $|\sigma|/\mathrm{k}$.

Therefore: Worst-case analysis is not so important in analyzing paging algorithm

- Can randomization help? Yes!!