CSE 521 - Algorithms

On-line Algorithms

Thanks to Tami Tamir

1

Introduction

Online Algorithms are algorithms that need to make decisions without full knowledge of the input. They have 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 optimum offline algorithm, the algorithm that has perfect knowledge of the future.

2

The Ski-Rental Problem

- Assume that you are taking ski lessons. After each lesson you decide (depending on how much you enjoy it, and what is your bones status) 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 y$.
- Will you buy or rent?

3

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 y times then buy before you start, otherwise always rent.
- The cost of this algorithm is min(t, y).

• This type of strategy, with perfect knowledge of the future, is known as an offline strategy.

4

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 = y guarantees that you never pay more than twice the cost of the offline strategy.
- Example: Assume y=7$ Thus, after 6 rents, you buy. Your total payment: 6+7=13$.

5

The Ski-Rental Problem

Theorem: Setting k = y 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, $t \geq y$.

- Your total payment is k-1+y =2y-1.
- The offline cost is min(t, y) = y.
- The ratio is (2y-1)/y = 2-1/y. ∎

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

6

Competitive Ratio

- An on-line algorithm A is c-competitive if there is a constant b for all sequences s of operations

  $$A(s) \leq c\ OPT(s) + b$$

  where A(s) is the cost of A on the sequence s and OPT(s) is the optimal off-line cost for the same sequence.
- Competitive ratio is a worst case bound.

7

The Ski-Rental Problem

Is there a better strategy?

- 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+y and the optimum offline cost is min(k,y).
- For every k, the ratio (k-1+y)/min(k,y) is at least (2-1/y)
- Therefore, every strategy is at least (2-1/y)- -competitive.

  ■

8

The Ski-Rental Problem

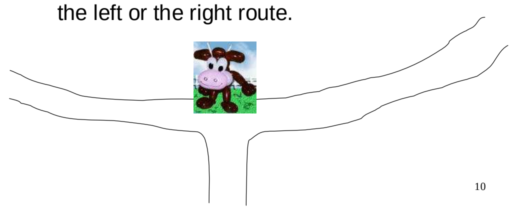
The general rule:

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.

9

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.



10

The Lost Cow Problem

Old McDonald's algorithm:

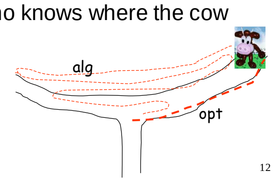
1. d=1; current side = right
2. repeat:
   i. Walk distance d on current side
   ii. if find cow then exit
   iii. else return to starting point
   iv. d = 2d
   v. Flip current side

11

The Lost Cow Problem

Theorem: Old McDonald's 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).



12

The Lost Cow Problem

Theorem: Old McDonald's 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?).

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

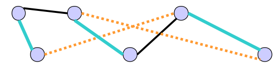
Cost OPT $= 2^j + \varepsilon > 2^j$

Cost ON $= 2(1 + 2 + 4 + \ldots + 2^{j+1}) + 2^j + \varepsilon$

$= 2 \cdot 2^{j+2} + 2^j + \varepsilon = 9 \cdot 2^j + \varepsilon < 9 \cdot$ Cost OPT ■

Edge Coloring

- An Edge-coloring of a graph $G=(V,E)$ is an assignment, c, of integers to the edges such that if e_1 and e_2 share an endpoint then $c(e_1) \neq c(e_2)$.



- Let Δ be the maximal degree of some vertex in G.
- In the offline case, it is possible to edge-color G using Δ or $\Delta+1$ colors (Δ colors are necessary and it is NP-hard to determine whether $\Delta+1$ are required).
- Online edge coloring: The graph is not known in advance. In each step a new edge is added and we need to color it before the next edge is known.

Optimal Online Algorithm for Edge Coloring

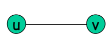
- We color the edges with numbers 1,2,3…

- Let $e=(u,v)$ be a new edge.

Color e with the smallest color which is not used by any edge adjacent to u or v.

Claim: The algorithm uses at most $2\Delta-1$ colors.

Proof outline (was hw6 q.1): assume we need the color 2Δ. It must be that all the colors $1,2,\ldots,2\Delta-1$ are used by edges adjacent to u or v. Therefore, either u or v has Δ adjacent edges, excluding e, contradicting the definition of Δ.

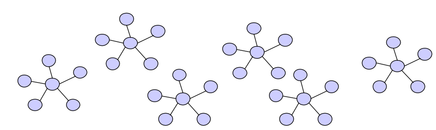


Online Edge Coloring

Claim: Any deterministic algorithm needs at least $2\Delta-1$ colors.

Proof: Assume there is an algorithm that uses only $2\Delta-2$ colors. Given Δ we add to the graph many $(\Delta-1)$-stars.

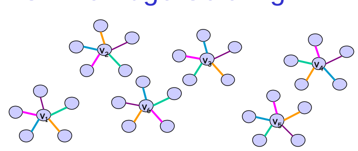
Example:
$\Delta=6$



There is a finite number of ways to edge-color a $(\Delta-1)$-star with colors from $\{1,2,\ldots,2\Delta-2\}$, so at some point we must have Δ stars, all colored with the the same set of $\Delta-1$ colors.

Online Edge Coloring

Δ stars, all colored with the the same set of Δ-1 colors.



Let $v_1, v_2, \ldots, v_\Delta$ be the centers of these stars.

We are ready to shock the algorithm!.

We add a new vertex, a, and Δ edges $(a\text{-}v_1), \ldots, (a,v_\Delta)$.

Each new edge must have a unique color (why?), that is not one of the $(\Delta-1)$ colors used to color the stars (why?) à $2\Delta-1$ colors must be used.

Note: the maximal degree is Δ

Online Scheduling and Load Balancing

Problem Statement:

- A set of m identical machines,
- A sequence of jobs with processing times $p_1, p_2, \ldots$.
- Each job must be assigned to one of the machines.
- When job j is scheduled, we don't 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 $= \max_i \sum_{j \text{ on } M_i} p_j$.

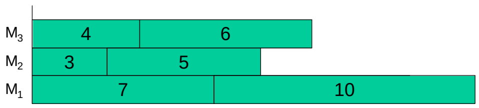
(the maximal load on one machine)

Online Scheduling and Load Balancing

List Scheduling [Graham 1966]:

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

Example: m=3 σ = 7 3 4 5 6 10



Makespan = 17

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}.

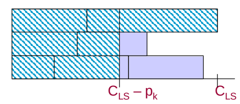
All the machines are busy when j starts its processing, thus, $\forall j,\ H_j \geq C_{LS} - p_k$.

For at least one machine (that processes k) $H_j = C_{LS}$.

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

à $\sum_i p_i + (m-1)p_k \geq mC_{LS}.$

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



Online Scheduling and Load Balancing

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

Consider an optimal offline schedule.

$C_{opt} \geq \max_i p_i \geq p_k$ (some machine must process the longest job).

$C_{opt} \geq 1/m \sum_i p_i$ (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.

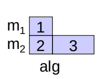
	deterministic			randomized	
m	lower bound	upper bound	LS	lower bound	upper bound
2	1.5	1.5	1.5	1.334	1.334
3	1.666	1.667	1.667	1.42	1.55
4	1.731	1.733	1.75	1.46	1.66
∞	1.852	1.923	2	1.58	---

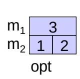
A lower Bound for Online Scheduling

Theorem: For m=2, no algorithm has r< 1.5

Proof: Consider the sequence σ = 1,1,2.

If the first two jobs are scheduled on different machines, the third job completes at time 3.

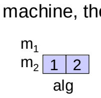


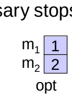


C_A=3, C_{opt}=2

r=3/2

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





C_A=2, 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 M1 that will be requested farthest out in the future.

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) ≤ 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(FIFO) ≥ 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 on-line 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 $|\sigma|$.

31

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 $k+1^{th}$ request or later)

•The total cost of LFD is at most $|\sigma|/k$.

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

•Can randomization help? Yes!!

32