6. Online Algorithms

—> Video

Offline Optimization: Input data of the problem instance is completely given, and based on this an algorithm computes an optimal (or approximate) solution.
Online Optimization: Many real-world problems need decisions made without knowledge about future events, that is, not the complete input data is given. That is, each of the decisions must be made based on past events without secure information about the future.

6.1 Examples for online problems:

- **The Ski Rental Problem:** A person goes skiing the first time in her life, she is unsure whether to rent or to buy skiis. She could rent skis for 50€ per day, or she could buy skis for 500€. What is the minimum-cost strategy?
  You do not know how often in the future she will go skiing/how long the good weather holds….
- **The BahnCard Problem:** In Germany there exists so called BahnCards (of different levels), the BahnCard50 costs 255€ for a year, and during this year, all tickets are half-price (50% of the original).
  Q: When to buy a BahnCard?
  Problem: You do not know about all travel within the next 12 months.
- **Online Search (The Cow ;)):** A cow searches for her hay, but does not know whether the haystack is to her right or to her left and in which distance.
  Q: How can the cow find the hay while walking as little as possible?

✤ **Paging:** A fundamental online optimization problem

Given: Two-level virtual memory system with fast memory (cache) for k pages, and the slow memory for N pages.

Slow memory: fixed set $P=\{p_1, p_2, \ldots, p_N\}$

Cache: can store any k-subset of P, $k<N$

Request: Access to a page in the memory system

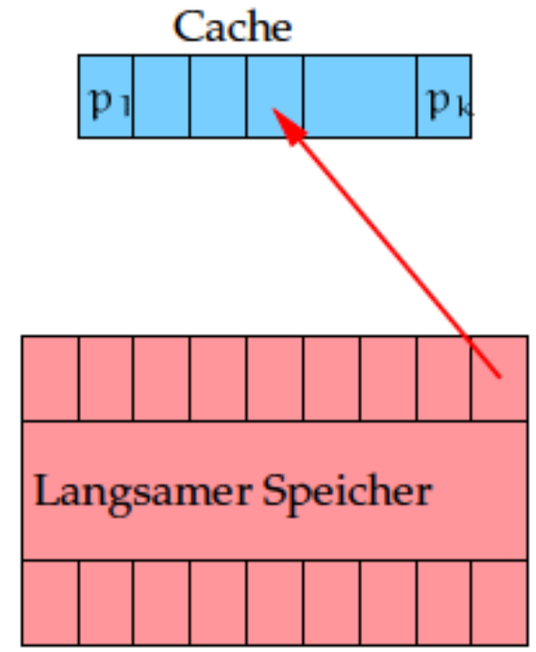
Request for page $p_i \Longrightarrow$ system must make p_i available in cache

p_i in the cache (hit), system must do nothing

p_i not in the cache (miss), system incurs page fault, and must copy the page p_i from slow memory to one of the locations in the cache

To do so: which page to evict from cache?

Goal: Minimize the number of page faults



↑ cost model: page fault model (charge 1 for bringing page to fast memory, 0 for reading from or writing a page in the fast memory

More accurate model: consider the ratio between time for fast memory access and the time to fetch a page from the slow memory into fast memory

→ full access cost model: charge 1 for each access to the fast memory, and s to move a page from slow to fast memory

image source: “Online Optimierung”, from Sven O. Krumke, Jörg Rambau, 12. April 2000

How do we evaluate how good an online algorithm is?
"Classical" worst-case analysis does usually not make sense
Example: for paging *each* algorithms has in the worst case a page fault at *every* request
Average-case analysis?
We would need statistical model for the input (usually not known)
$\Longrightarrow$ Sleator and Tarjan suggested the **competitive analysis** [D. D. Sleator and R. E. Tarjan, Amortized efficiency of list update and paging rules , Communications of the ACM 28 (1985), no. 2, 202–208]
We measure the quality of an online algorithm by comparing its performance to that of an optimal offline algorithm (which for an online problem is an unrealisable algorithm with full knowledge of the future).

Definition 6.2 (Competitive analysis):
Let σ be a feasible input for an optimisation problem (maximization or minimization)
Let ALG be the online algorithm, with cost $C_{ALG}(\sigma)$
Let OPT be the optimal online algorithm, which knows the sequence σ a priori, with cost $C_{OPT}(\sigma)$
ALG is **c-competitive** if $\exists\alpha$: $C_{ALG}(\sigma) \leq c\, C_{OPT}(\sigma) + \alpha$ for all finite input sequences σ
When for the additive constant $\alpha \leq 0$, we can say ALG is strictly c-competitive
ALG attains a **competitive ratio** c
The infinum over the set of all values c such that ALG is c-competitive is called the **competitive ratio** of ALG, $\mathcal{R}$(ALG).

c may be function of the problem parameters, but must be independent of input σ.
Example: scheduling problem with N machines —> c may depend on N, but must be independent of type and number of jobs being scheduled.

(Strictly) c-competitive online algorithm ALG: a c-approximation with the restriction that ALG must compute online.
$\Longrightarrow$ For each input σ a c-competitive algorithm is guaranteed to incur a cost within a factor c of the optimal offline cost (up to additive constant α)

Why allow positive α?

- For intrinsic online problems, like paging, we have an arbitrarily long input sequence w/ unbounded cost
- Longer and longer subsequences $\Longrightarrow$ α becomes insignificant
- For finite input sequences: use of additive constant α allows for an intrinsic performance ratio not dependent on initial conditions
- For “bounded cost” opt problems (e.g., graph colouring with at most N colors) more significant how α depends on N

How to view online algorithms?
Often: **online player and a malicious adversary**
adversary creates input
online player runs an online algorithm on that input
adversary, based on knowledge of the algorithm used by the online player, constructs the worst possible input so as to maximise the competitive ratio
sometimes: adversary + optimal offline algorithm = **offline player**
For deterministic online algorithms: adversary knows exactly what the input player's response will be to each input element
The adversary is **omniscient**

6.1 Paging

Deterministic paging algorithms:

- **LRU** (Least Recently Used): Evict page that was requested least recently (that is, whose most recent request was earliest).
- **FIFO** (First In First Out): Evict page that has been in fast memory longest.
- **LIFO** (Last In First Out): Evict page that most recently moved to the fast memory.
- **LFU** (Least Frequently Used): Evict page that has been requested least frequently since entering the fast memory.
- **LFD** (Longest Forward Distance): Evict page whose next request is latest (furthest in the future).
- ↑LFD is online algorithm! (requires full knowledge of future requests)
- All are **demand paging**, that is, unless there is a page fault, they never evict a page from the cache.

Theorem 6.3: LFD is an optimal offline algorithm for paging.

Proof: We show how any optimal offline paging algorithm can be modified to act like LFD without degrading its performance

Claim 6.4: *Let ALG be any paging algorithm. Let σ be any request sequence. For any i, i=1,2,…,|σ|, it is possible to construct an offline algorithm ALG_i that satisfies three properties:*
(i) ALG_i processes the first i-1 requests exactly as ALG does
(ii) If the ith request results in a page fault, ALG_i evicts from its fast memory the page with the "longest forward distance"
(iii) $ALG_i(\sigma) \leq ALG(\sigma)$

Claim established $\Longrightarrow$ apply it n=|σ| times:

Fix any request sequence σ.
Starting with any optimal offline algorithm OPT, we apply the claim with i=1 to obtain OPT_1, then apply the claim to algorithm OPT_1 with i=2 to obtain OPT_2, and so on.
OPT_n acts identically to LFD w.r.t. σ

Theorem 6.3: LFD is an optimal offline algorithm for paging.

Proof: We show how any optimal offline paging algorithm can be modified to act like LFD without degrading its performance

Claim 6.4: *Let ALG be any paging algorithm. Let σ be any request sequence. For any i, i=1,2,…,|σ|, it is possible to construct an offline algorithm ALG_i that satisfies three properties:*
(i) ALG_i processes the first i-1 requests exactly as ALG does
(ii) If the ith request results in a page fault, ALG_i evicts from its fast memory the page with the "longest forward distance"
(iii) $ALG_i(\sigma) \leq ALG(\sigma)$

Proof of Claim 6.4:
Given ALG, we construct algorithm ALG_i.
For set of pages X and a page p, we denote by $X+p=X \cup \{p\}$
Assume, immediately after processing ith request the fast memories of ALG and ALG_i contain the page sets X+v, X+u, respectively (With X having k-1 (common) pages, and v and u are any pages).
W.l.o.g. we assume $v \neq u$ ($\Longrightarrow$ith request resulted in a page fault)
Until v is requested, ALG_i mimics ALG except for evicting u if ALG evicts v
This is feasible: after servicing any reuest in this fashion, the number of common pages remains at least k-1
+ if at any time the number of common pages becomes k (i.e., if ALG evicts v), ALG_i identifies with ALG from that point onward and the proof is complete.

Theorem 6.3: LFD is an optimal offline algorithm for paging.

Proof: We show how any optimal offline paging algorithm can be modified to act like LFD without degrading its performance

Claim 6.4: *Let ALG be any paging algorithm. Let σ be any request sequence. For any i, i=1,2,…,|σ|, it is possible to construct an offline algorithm ALG_i that satisfies three properties:*
(i) ALG_i processes the first i-1 requests exactly as ALG does
(ii) If the ith request results in a page fault, ALG_i evicts from its fast memory the page with the "longest forward distance"
(iii) $ALG_i(\sigma) \leq ALG(\sigma)$

Proof of Claim 6.4 ctd.:
If v is eventually requested and ALG and ALG_i have not yet been identified, this will incur a page fault to ALG_i but not to ALG.
But: by the time v is requested it had the longest forward distance
$\Longrightarrow$ there must have been at least one request for u after the ith page fault

The first such request incurs a page fault to ALG but not to ALG_i
$\Longrightarrow$ total number of page faults for ALG_i after servicing v is equal to that for ALG

+In order to serve the request for v, ALG_i evicts u
$\Longrightarrow$ The two algorithms identify

A natural generalisation of the paging problem:

Definition 6.5: The (h,k)-Paging Problem

Let k and h be positive integers with $h \leq k$

In the (h,k)-paging problem, we measure the performance of an online paging algorithm with a cache of size k relative to an optimal offline paging algorithm with cache of size $h \leq k$

$\Longrightarrow$ If $h<k$ we provide the optimal offline algorithm with strictly less resources.

Some natural algorithms incur **Belady's anomaly**: on some input sequences the algorithm may perform better when it has a smaller fast memory.

Easy to see: optimal algorithm does not incur Belady's anomaly (since it does not need to use all its cache pages).

$\Longrightarrow$ Homework 6, problem 4

Why interesting?

Reducing power seems "unfair"

But: in competitive analysis we give offline player unrealistic powers

Parameterize size of the optimal offline cache separately

$\Longrightarrow$ We can measure strength of online paging algorithms agains weaker adversaries, where the power of the adversary is quantitatively controlled

Optimal competitive ratio?
One more algorithm, not a demand paging algorithm.
FWF (Flush When Full): Whenever there is a page fault and there is not space left in the cache, evict all pages currently in the cache (call this action a “flush”).

Can be upgraded to be demand paging:
instead of flushing the cache, it can mark all flushed pages, and whenever there is a new page fault, it can accommodate new page by evicting an arbitrary marked page.

We show: LRU, FIFO and FWF all attain optimal competitive ratio.

Plan:

- Define “marking algorithms”
- Prove that every marking algorithm attains a competitive ratio of k/(k-h+1) for the (h,k)-paging problem
  - Gives an upper bound of k for the basic paging problem ((k,k,)-paging)
- Define “conservative algorithms”: provide similar but different general class of algorithms that achieve the same k/(k-h+1)-competitive ratio
- Give a lower bound of k on the competitive ratio of any deterministic online paging algorithm
  - Can be generalised to the (h,k)-paging problem
  - All the above algorithms are optimal for (h,k)-paging

Marking algorithms

Fast memory of size k

Fixed request sequence σ

We divide the request sequence into **phases**: phase 0 is the empty sequence.

For every i≥1, phase i is the maximal sequence following phase i-1 that contains at most k distinct page requests

$\Longrightarrow$ (if exists) phase k+1 begins on the request that constitutes the (k+1)st distinct page request since the start of the i-th phase

= k-phase partition (independent of how any particular all processes σ)

Let σ be any request sequence, and consider its k-phase partitions

With each page of the slow memory associate a bit called its **mark**

- When mark bit is set, page is marked
- ow it is unmarked

Suppose: at beginning of each k-phase, we unmark all the pages currently in the fast memory

During a k-phase, we mark a page when it is first requested during the k-phase.

A **marking algorithm** never evicts a marked page from its fast memory.

We can consider FWF as the most elementary marking algorithm.

Upper bound on the competitiveness of any online marking algorithm:
Theorem 6.6: Let ALG be any marking algorithm with a cache of size k, and let OPT be any optimal offline algorithm with a cache of size h≤k. Then ALG is k/(k-h+1)-competitive.

Proof does not use any property of OPT (except that paging algorithm with a cache of size h).
Proof: Fix request sequence σ and consider its k-phase partition.
First claim: for any phase i≥1, a marking algorithm ALG incurs at most k page faults.
k distinct page references in each phase (except for, possibly, the last phase)
Once page accessed, it is marked, and therefore cannot be evicted until the phase has been completed
$\Longrightarrow$ ALG cannot fault twice on the same page

For any i≥1, let q be the first request of phase i
consider the input sequence starting with the second request of phase i up to and including the first request of phase i+1 (assuming phase i+1 exists)
Algorithm OPT has h-1 pages not including q
There are k requests in this sequence not counting q
$\Longrightarrow$ OPT must incur at least k-(h-1)=k-h+1 faults to service this sequence of requests
If phase i is the last phase: OPT pays at least k'-h+1, with k'=number of distinct page requests during the last phase. We will ignore this cost to OPT
During each phase ALG faults at most k times
For each phase (except last) OPT has at least k-h+1 faults

$\Longrightarrow \mathrm{ALG}(\sigma) \leq \frac{k}{k-h+1} \times \mathrm{OPT}(\sigma) + \alpha$ (α≤k: max #page faults by ALG in last phase)

Proof as homework:
Lemma 6.7: LRU is a marking algorithm.

Corollary 6.8: LRU is k/(h-k+1)-competitive.

Definition 6.9: We say that a paging algorithm is **conservative** if it satisfies the following condition: *On any consecutive input subsequence containing k or fewer distinct page references, ALG will incur k or fewer page faults*.

Modifying the proof of Theorem 6.6, we obtain:
Theorem 6.10: Let ALG be any online conservative paging algorithm with a cache of size k, and let OPT be any optimal offline algorithm with a cache of size $h \leq k$. Then ALG is k/(k-h+1)-competitive.

Here without proof:
Theorem 6.11: let ALG be any paging algorithm, then $\mathcal{R}(ALG) \geq k$.