Chapter 8
Online Algorithms

Algorithm Theory
WS 2012/13

Fabian Kuhn
Online Computations

- Sometimes, an algorithm has to start processing the input before the complete input is known

- For example, when storing data in a data structure, the sequence of operations on the data structure is not known

**Online Algorithm:** An algorithm that has to produce the output step-by-step when new parts of the input become available.

**Offline Algorithm:** An algorithm that has access to the whole input before computing the output.

- Some problems are inherently online
  - Especially when real-time requests have to be processed over a significant period of time
Competitive Ratio

• Let’s again consider optimization problems
  – For simplicity, assume, we have a minimization problem

Optimal offline solution \( \text{OPT}(I) \):
• Best objective value that an offline algorithm can achieve for a given input sequence \( I \)

Online solution \( \text{ALG}(I) \):
• Objective value achieved by an online algorithm \( \text{ALG} \) on \( I \)

**Competitive Ratio:** An algorithm has competitive ratio \( c \geq 1 \) if
\[
\text{ALG}(I) \leq c \cdot \text{OPT}(I) + \alpha.
\]
• If \( \alpha \leq 0 \), we say that \( \text{ALG} \) is strictly \( c \)-competitive.
Paging Algorithm

Assume a simple memory hierarchy:

- Page in fast memory (hit): take page from there
- Page not fast memory (miss): leads to a page fault
- Page fulla: the page is loaded into the fast memory and some page has to be evicted from the fast memory
- Paging algorithm: decides which page to evict
- Classical online problem: we don’t know the future accesses
Paging Strategies

Least Recently Used (LRU):
• Replace the page that hasn’t been used for the longest time

First In First Out (FIFO):
• Replace the page that has been in the fast memory longest

Last In First Out (LIFO):
• Replace the page most recently moved to fast memory

Least Frequently Used (LFU):
• Replace the page that has been used the least

Longest Forward Distance (LFD):
• Replace the page whose next request is latest (in the future)
• LFD is not an online strategy!
LFD is Optimal

**Theorem:** LFD (longest forward distance) is an optimal offline alg.

**Proof:**

- For contradiction, assume that LFD is not optimal
- Then there exists a finite input sequence $\sigma$ on which LFD is not optimal (assume that the length of $\sigma$ is $|\sigma| = n$)
- Let OPT be an optimal solution for $\sigma$ such that
  - OPT processes requests $1, \ldots, i$ in exactly the same way as LFD
  - OPT processes request $i + 1$ differently than LFD
  - Any other optimal strategy processes one of the first $i + 1$ requests differently than LDF
- Hence, OPT is the optimal solution that behaves in the same way as LFD for as long as possible $\rightarrow$ we have $i < n$
- Goal: Construct $OPT'$ that is identical with LFD for req. $1, \ldots, i + 1$
LFD is Optimal

**Theorem:** LFD (longest forward distance) is an optimal offline alg.

**Proof:**

**Case 1:** Request $i + 1$ does **not** lead to a page fault

- LFD does not change the content of the fast memory
- OPT behaves differently than LFD
  - As up to request $i + 1$, both algorithms behave in the same way, they also have the same fast memory content
  - OPT therefore does not require the new page for request $i + 1$
  - Hence, OPT can also load that page later (without extra cost) → OPT’
LFD is Optimal

**Theorem:** LFD (longest forward distance) is an optimal offline alg.

**Proof:**

**Case 2:** Request $i + 1$ does lead to a page fault

- LFD and OPT move the same page into the fast memory, but they evict different pages
  - If OPT loads more than one page, all pages that are not required for request $i + 1$ can also be loaded later
- Say, LFD evicts page $p$ and OPT evicts page $p'$
- By the definition of LFD, $p'$ is required again before page $p$
LFD is Optimal

**Theorem:** LFD (longest forward distance) is an optimal offline alg.

**Proof:**

**Case 2:** Request \( i + 1 \) does lead to a page fault

\[ i + 1 \]

LFD evicts \( p \)

OPT evicts \( p' \)

\[ \ell' < \ell: \text{OPT evicts } p \]

\[ j': \text{next req. for } p' \]

\[ j: \text{next req. for } p \]

\[ \ell \leq j': \text{OPT loads } p' \text{ (for first time after } i + 1) \]

a) OPT keeps \( p \) in fast memory until request \( \ell \)
   
   Evict \( p \) at request \( i + 1 \), keep \( p' \) instead and load \( p \) (instead of \( p' \)) back into the fast memory at request \( \ell \)

b) OPT evicts \( p \) at request \( \ell' < \ell \)

   Evict \( p \) at request \( i + 1 \) and \( p' \) at request \( \ell' \) (switch evictions of \( p \) and \( p' \))
Phase Partition

We partition a given request sequence $\sigma$ into phases as follows:

- **Phase 0**: empty sequence
- **Phase $i$**: maximal sequence that immediately follows phase $i - 1$ and contains at most $k$ distinct page requests

**Example sequence ($k = 4$):**

\[
2, 5, 12, 5, 4, 2, 10, 8, 3, 6, 2, 2, 6, 6, 8, 3, 2, 6, 9, 10, 6, 3, 10, 2, 1, 3, 5
\]

**Phase $i$ Interval**: interval starting with the second request of phase $i$ and ending with the first request of phase $i + 1$

- If the last phase is phase $p$, phase-interval $i$ is defined for $i = 1, \ldots, p - 1$
Optimal Algorithm

Lemma: Algorithm LFD has at least one page fault in each phase interval (for \( i = 1, \ldots, p - 1 \), where \( p \) is the number of phases).

Proof:

- \( q \) is in fast memory after first request of phase \( i \)
- Number of distinct requests in phase \( i \): \( k \)
- By maximality of phase \( i \): \( q' \) does not occur in phase \( i \)
- Number of distinct requests \( \neq q \) in phase interval \( i: k \)

\( \rightarrow \) at least one page fault
LRU and FIFO Algorithms

**Lemma:** Algorithm LFD has at least one page fault in each phase interval $i$ (for $i = 1, \ldots, p - 1$, where $p$ is the number of phases).

**Corollary:** The number of page faults of an optimal offline algorithm is at least $p - 1$, where $p$ is the number of phases.

**Theorem:** The LRU and the FIFO algorithms both have a competitive ratio of at most $k$.

**Proof:**

- In phase $i$ only pages from phases before phase $i$ are evicted from the fast memory $\rightarrow \leq k$ page faults per phase
  - As long as not all $k$ pages from phase $i$ have been requested, the least recently used and the first inserted are from phases before $i$
  - When all $k$ pages have been requested, the $k$ pages of phase $i$ are in fast memory and there are no more page faults in phase $i$
Lower Bound

**Theorem:** Even if the slow memory contains only \( k + 1 \) pages, any deterministic algorithm has competitive ratio at least \( k \).

**Proof:**

- Consider some given deterministic algorithm ALG
- Because ALG is deterministic, the content of the fast memory after the first \( i \) requests is determined by the first \( i \) requests.
- Construct a request sequence inductively as follows:
  - Assume some initial slow memory content
  - The \((i + 1)^{st}\) request is for the page which is not in fast memory after the first \( i \) requests (throughout we only use \( k + 1 \) different pages)
- There is a page fault for every request
- OPT has a page fault at most every \( k \) requests
  - There is always a page that is not required for the next \( k - 1 \) requests
Randomized Algorithms

• We have seen that deterministic paging algorithms cannot be better than $k$-competitive

• Does it help to use randomization?

**Competitive Ratio:** A randomized online algorithm has competitive ratio $c \geq 1$ if for all inputs $I$,

$$\mathbb{E}\left[\text{ALG}(I)\right] \leq c \cdot \text{OPT}(I) + \alpha.$$

• If $\alpha \leq 0$, we say that ALG is strictly $c$-competitive.
Adversaries

• For randomized algorithm, we need to distinguish between different kinds of adversaries (providing the input)

Oblivious Adversary:
• Has to determine the complete input sequence before the algorithm starts
  – The adversary cannot adapt to random decisions of the algorithm

Adaptive Adversary:
• The adversary knows how the algorithm reacted to earlier inputs
• online adaptive: adversary has no access to the randomness used to react to the current input
• offline adaptive: adversary knows the random bits used by the algorithm to serve the current input
Lower Bound

The adversaries can be ordered according to their strength:

\[ \text{oblivious} < \text{online adaptive} < \text{offline adaptive} \]

- For algorithm that works against an oblivious adversary, also works with and online/offline adaptive adversary.
- A lower bound that holds against an offline adaptive adversary also holds for the other 2.
- ...

**Theorem:** No randomized paging algorithm can be better than \(k\)-competitive against an online (or offline) adaptive adversary.

**Proof:** The same proof as for deterministic algorithms works.

- Are there better algorithms with an oblivious adversary?