Deterministic paging algorithms

You have a fast memory (cache) of size k and a large slow memory. Given a sequence of page requests, serving a request r is done at cost 0 of the page is in the cache and at cost 1 if the page is not there: it has to be brought into the cache. The goal is to design an algorithm to minimize the number of page faults (requests that incur a cost). The algorithm chooses which page to evict from the cache to make room for the current request. The algorithm is online is its choice is done without knowledge of future requests.

Some popular online heuristics: LRU (Least recently used), FIFO (First in first out), CLOCK (1-bit version of LRU), LIFO (Last in first out), LFU (Least frequently used). One well-known offline heuristic: LFD (Longest forward distance).

Theorem 1 *LFD is optimal.*

Fix a sequence of n page requests. For all i, let $C_0^{(i)}$ denote the state of the cache after request i is served for the some particular sequence of page evictions, and let $C_0 = (C_0^{(i)})_{1 \leq i \leq n}$.

Let $C_t = (C_t^{(i)})_i$ be defined inductively as follows: up to and including time t, use LFD to choose which page to evict when there is a page fault. This determines the state of the cache at each timestep until time t. Then, follow the choice of evictions of C_{t-1} for all times $> t$ except one defined as follows: assume that at time t algorithm C_{t-1} evicts page u whereas LFD evicts page v. Let t' be the first time when, either C_{t-1} evicts page v, or C_{t-1} brings page u back into the cache (by evicting some page u'), whichever comes first. In the first case, at time t' C_t will evict page u, and in the second case, at time t' C_t will evict page u'. Either way, after serving request at time t' the state of the cache is the same for C_{t-1} and for C_t. By definition of LFD, between time t and time t' there can be no request for page v, so C_t is well-defined and incurs no more pages faults than C_{t-1}.

By induction C_t is at least as good as C_0. For $t = n$, C_n is exactly LFD, so LFD is at least as good as C_0. This was for an arbitrary C_0, so LFD is optimal.

The reader may wish to try this construction on an example, say for $k = 3$ and $n = 15$.

Definition 1 *An online algorithm ALG is c-competitive if there exists a b such that for every n and every sequence $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$, $ALG(\sigma) \leq cOPT(\sigma) + b$, where $ALG(\sigma)$ is the cost of ALG on sequence σ and $OPT(\sigma)$ is the optimal offline cost on σ.*

Theorem 2 *LRU is k-competitive.*

Fix a sequence of n pages requests. Decompose into phases as follows. Phase 1 starts with request σ_1. A phase starting with requests σ_i ends just before request σ_j such that the set of pages requested at times $\{i, \ldots, j-1\}$ has size k but the set of pages requested at times $\{i, \ldots, j\}$ has size $k+1$. In each phase, it is easy to see that LRU pays at most k pages faults. For each phase $\{i \ldots, j-1\}$, observing that the cache must contain page σ_i right after serving request i, it is easy to infer that OPT has to incur at least one page fault to serve requests $\{i+1, \ldots, j\}$. Hence the theorem.