Albert-Ludwigs-Universität, Inst. für Informatik
Prof. Dr. Fabian Kuhn
M. Ahmadi, A. R. Molla, O. Saukh February 22, 2016

Algorithm Theory, Winter Term 2015/16
Problem Set 13 - Sample Solution

Exercise 1: LRU with Potential Function (5 points)

When studying online algorithm, the total (average) cost for serving a sequence of requests can often be analyzed using amortized analysis. In the following, we will apply this to the paging problem and you will analyze the competitive ratio of the LRU algorithm by using a *potential function*. Recall that a potential function assigns a non-negative real value to each system state. In the context of online algorithms, we think of running an optimal offline algorithm and an online algorithm side by side and the system state is given by the combined states of both algorithms.
Consider the LRU paging algorithm, i.e., the online paging algorithm that always replaces the page that has been used least recently. Let $\sigma = (\sigma(1), \sigma(2), \ldots, \sigma(m))$ be an arbitrary request sequence of pages. Let OPT be some optimal offline algorithm. You can assume that OPT evicts at most one page in each step (e.g., think of OPT as the LFD algorithm). At any time t (i.e., after serving requests $\sigma(1), \ldots, \sigma(t)$), let $S_{\mathrm{LRU}}(t)$ be the set of pages in LRU's fast memory and let $S_{\mathrm{OPT}}(t)$ be the set of pages contained in OPT's fast memory. We define $S(t) := S_{\mathrm{LRU}}(t) \setminus S_{\mathrm{OPT}}(t)$. At each time t, we further assign an integer weights $w(p,t)$ from the range $\{1, \ldots, k\}$ to each page $p \in S_{\mathrm{LRU}}(t)$ such that for any two pages $p, q \in S_{\mathrm{LRU}}(t), w(p,t) < w(q,t)$ iff the last request to p occurred before the last request to q (i.e., the requests in S_{LRU} are numbered from $1, \ldots, k$ according to times of their last occurrences). Recall that we use k to denote the size of the fast memory. We define the potential function at time t to be

$$\Phi(t) := \sum_{p \in S(t)} w(p,t).$$

We define the amortized cost $a_{\mathrm{LRU}}(t)$ for serving request $\sigma(t)$ as

$$a_{\mathrm{LRU}}(t) := c_{\mathrm{LRU}}(t) + \Phi(t) - \Phi(t-1),$$

where $c_{\mathrm{LRU}}(t)$ is the actual cost for serving request $\sigma(t)$. Note that $c_{\mathrm{LRU}}(t) = 1$ if a page fault for algorithm LRU occurs when serving request $\sigma(t)$ and $c_{\mathrm{LRU}}(t) = 0$ otherwise. Similarly, we define $c_{\mathrm{OPT}}(t)$ to be the actual cost of the optimal offline algorithm for serving request $\sigma(t)$. Again, $c_{\mathrm{OPT}}(t) = 1$ if OPT encounters a page fault in step t and $c_{\mathrm{OPT}}(t) = 0$ otherwise. In order to show that the competitive ratio of the algorithm is at most k, you need to show that for every request $\sigma(t)$,

$$a_{\mathrm{LRU}}(t) \leq k \cdot c_{\mathrm{OPT}}(t).$$

Solution

To show that the LRU algorithm is k-competitive, it is required to show that, for all t

$$c_{\mathrm{LRU}}(t) + \Phi(t) - \Phi(t-1) \leq k \cdot c_{\mathrm{OPT}}(t) \tag{1}$$

Let $\sigma = (\sigma(1), \sigma(2), \ldots, \sigma(m))$ be an arbitrary sequence of requests. Consider an arbitrary request $\sigma(t) = p$ and W.l.o.g. assume that OPT serves the request earlier than LRU algorithm.

In case both algorithms OPT and LRU have no page fault on $\sigma(t) = p$, inequality (1) holds. Potential function can not increase (thus, it can decrease) and costs of both operations are zero.
If OPT does not have a page fault on $\sigma(t) = p$, then $c_{\text{OPT}} = 0$ and the potential function does not change. If OPT does have a page fault, $c_{\text{OPT}} = 1$ and it needs to evict a page. If the page we evict is not in the LRU's fast memory, the potential function does not change. And if this page is in the LRU's fast memory, the set S gets an additional element and the potential function increases by at most k.
Now, if we consider the LRU algorithm, if it does not have a page fault on $\sigma(t)$, then $c_{\text{LRU}} = 0$ and potential function does not change. If LRU has a page fault, $c_{\text{LRU}} = 1$ and we show that the value of potential function decreases by at least 1. In fact, before LRU serves the $\sigma(t) = p$ request, p is only in OPT's fast memory. By symmetry, there must be a page that is only in LRU's fast memory, so there has to exist a page $q \in S$. If q is evicted, then its weight is at least 1. Hence, potential function decreases by at least 1. Otherwise, if another page gets evicted, and p is loaded into LRU's fast memory, p gets weight at most k and all other pages in the set S including q decrease their weights by 1, so the potential function decreases by 1.
In case OPT and LRU have page fault and both need to evict a page, concluding the changes of cases described above, the costs are 1 and potential function increases by at most $k-1$ (increases by at most k due to the page fault of OPT and decreases by at least 1 due to page fault of LRU). Inequality (1) still holds: $1 + k - 1 \leq k \cdot 1$
To conclude, every time OPT has a fault, the potential function increases by at most k and every time LRU has a fault, the potential function decreases by at least 1 and the inequality (1) always holds.

Exercise 2: (Generalized) Online Load Balancing (7 points)

In the lecture, a simple greedy algorithm was presented for an instance of the load balancing problem where the *speeds* of all machines are assumed to be equal. This algorithm gives a factor 2 approximation for the minimum makespan. As we have seen, it is not necessary to sort the jobs at the beginning of the algorithm and we can always arbitrarily pick up a job and assign it to the machine with the smallest load. Hence, the greedy algorithm can also be used as an online algorithm for the corresponding online load balancing problem where the jobs arrive in an online fashion one by one. The algorithm then achieves a competitive ratio of at most 2.

Now we are interested to find an online algorithm with constant competitive ratio for a generalization of the above online load balancing problem where the *speeds* of machines are different; as before jobs arrive one at a time in an online fashion. Assume that there are m machines and that for a given job j, $t_i(j)$ is the processing time when running job j on machine i. For each $1 \leq i \leq m$ and $1 \leq j \leq n$, we assume that

$$t_i(j) := \frac{w(j)}{v(i)},$$

where *weight* $w(j)$ depends only on job j and *speed* $v(i)$ depends only on the machine i.

Consider the following procedure we run after job j arrives: assume that the machines are indexed according to **increasing speed**. Let $\vec{T}(j) := (T_1(j), T_2(j), \ldots, T_m(j))$ denote the loads of the m machines after job j is assigned.
Hence, in `Assign(`$\vec{T}$`, `λ`)`, job j is assigned to the slowest machine such that the load on this machine is still below 2λ after the assignment.

(a) (4 points) Show that if $\lambda \geq T^*$ (where T^* is the optimal makespan), the procedure `Assign` always succeeds (i.e., it never returns **fail**). As a consequence, if we knew the optimal makespan T^*, choosing $\lambda = T^*$ would lead to a competitive ratio of 2.

Hint: *By way of contradiction, assume that some job t cannot be assigned. Then show that there must exist some job s (which was assigned earlier than t) which could have been assigned to a slower machine.*

Procedure Assign($\vec{T}(j-1), \lambda$)

```
/* λ is a given parameter.
Let S := {i : T_i(j−1) + t_i(j) ≤ 2λ};
if S = ∅ then
    b := fail
else
    k := min{i : i ∈ S};
    T_k(j) := T_k(j−1) + t_k(j);
    b := success
end
return (T(j), b).
```

(b) (3 points) Using the result of (a), devise an online algorithm with $\mathcal{O}(1)$ competitive ratio (for the case when the optimal makespan is not known). using `Assign` repeatedly (with changing λ) for solving an instance of online load balancing where the speed of machines are not necessarily equal.

Hint: *Use `Assign` repeatedly (with changing λ). Define phases such that at the beginning of the first phase, λ is set to be equal to the load (processing time) generated by the first arrival job on the fastest machine.*

Remark: *Note that if you could not solve (a), you can still try to solve (b).*

Solution

(a) Assume that `Assign`($\vec{T}$, λ), where $T^* \leq \lambda$, fails on task j. Let r be the fastest machine whose load does not exceed T^*, so $r = \max\{i \,|\, T_i(j-1) \leq T^*\}$. If there is no such machine, set $r = 0$. It is clear that $r \neq m$, otherwise, j could have been assigned to the fastest machine m, since $T_m(j-1) + t_m(j) \leq T^* + T^* \leq 2\lambda$. Define $S = \{i \,|\, i > r\}$, the set of the *overloaded* machines. Since $r < m$, S is not empty. Denote J_i and J_i^* sets of jobs assigned to machine i by online and offline algorithms, respectively. Since we are dealing with machines that have different speed, we have:

$$\sum_{i\in S, s\in J_i} t_m(s) = \sum_{i\in S, s\in J_i} \frac{t_m(s)}{t_i(s)} t_i(s) = \sum_{i\in S} \frac{v_i}{v_m} \sum_{s\in J_i} t_i(s) > \sum_{i\in S} \frac{v_i}{v_m} T^* \geq \sum_{i\in S} \frac{v_i}{v_m} \sum_{s\in J_i^*} t_i(s) = \sum_{i\in S, s\in J_i^*} t_m(s)$$

This implies that there exists a job $s \in \cup_{i\in S} J_i$, such that $s \notin \cup_{i\in S} J_i^*$, that is, there exists a job assigned by the online algorithm to a machine $i \in S$, and assigned by the offline algorithm to a slower machine $i' \notin S$.

By our assumption $t_{i'}(s) \leq T^* \leq \lambda$. Since $r \geq i'$, machine r is at least as fast as machine i', and thus $t_r(s) \leq T^* \leq \lambda$. Since job s was assigned before job j, $T_r(s-1) \leq T_r(j-1) \leq T^*$. But this means that the online algorithm should have placed job s on r or a slower machine instead of i, which is a contradiction. Hence, `Assign` never returns fail.

(b) Consider the following algorithm, that reuses the : `Assign`($\vec{T}$, λ) algorithm. The algorithm consists of phases, for each of those we define parameter λ_n and run the `Assign`($\vec{T}$,λ_n) until it returns *fail*. λ_i is defined in the following way: $\lambda_0 = 0$. After we assign the first job to the fastest machine at the beginning of the first phase, we set λ_1 to be the load generated by the first job on the fastest machine for this job. At the beginning of any phase $h > 1$ we set $\lambda_h = 2 \cdot \lambda_{h-1}$. During any phase $h > 1$ jobs are assigned independently of the jobs assigned in the previous phases. This means that each next phase starting with loads zero of the machines. Phase h ends, when algorithm returns *fail*. Then we start the next phase with doubling the λ-value. This approach can increase the competitive factor by at most 4 (a factor of 2 due to the load in all the rest of the phases except the last one, and another factor 2 due to imprecise approximation of λ). This

is easy to see. We double λ for every phase until we find such a value that is at least as large as the optimal *makespan* (we remember from the question (a), that if $T^* \leq \lambda$ the `Assign(`$\vec{T}$`, `λ`)` does not *fail*), but this imprecise way of searching for the appropriate λ might give us the value that is twice larger than we need. Another factor of 2 coming because the phases are considered to be independent, so we do not take into account the loads of the machines from previous phases while assigning the jobs in the current phase. If we sum up the loads that we got in each phase we can see that for every machine the total load is at most twice as large as when we have not treated the phases independently. Since the designed performance guarantee of `Assign(`$\vec{T}$`, `λ`)` is 2, we get competitive ration of 8, which is constant.