University of Freiburg
Dept. of Computer Science
Prof. Dr. F. Kuhn
M. Ahmadi, P. Schneider

# Algorithms Theory
# Sample Solution Exercise Sheet 8

**Due:** Monday, 18th of February, 2019, 14:15 pm

*This is a bonus exercise. You can earn points as usual but the threshold for exam admittance remains unchanged.*

## Exercise 1: Minimum Vertex Cover vs Maximum Clique *(2+4 Points)*

A vertex cover of a graph $G = (V, E)$ is a set $V' \subseteq V$ of nodes such that for all edges $\{u, v\} \in E$ we have $\{u, v\} \cap V' \neq \emptyset$. The minimum vertex cover problem requires to find a vertex cover of minimum cardinality.
A clique of graph $G = (V, E)$ is a subset $V' \subseteq V$ of nodes such that for all $u, v \in V'$ it holds that $\{u, v\} \in E$. The maximum clique problem requires to find a clique of maximum cardinality. Let us define the complement of graph $G = (V, E)$ as $\bar{G} = (V, \bar{E})$, where

$$\bar{E} = \big\{\{u, v\} \mid u, v \in V \text{ and } \{u, v\} \notin E\big\}.$$

(a) Given a minimum vertex cover for the complement $\bar{G}$ of graph $G$, explain how one can achieve a maximum clique of $G$.

Let us assume that we are given a 2-approximation algorithm $\mathcal{A}$ for the minimum vertex cover problem. Consider the following algorithm, denoted by $\mathcal{B}$, which uses $\mathcal{A}$ as its subroutine to compute a clique in graph $G$: It runs $\mathcal{A}$ on $\bar{G}$ and then uses the same technique as in part (a) to get a vertex cover $V'$.

(b) Argue why the approximation ratio of $\mathcal{B}$ could be super-constant.

## Sample Solution

(a) Having a minimum vertex cover $C$ of $\bar{G}$, one returns $V \setminus C$ as a maximum clique of $G$.

Now let us explain why $V \setminus C$ is a maximum clique of $G$. To do so, it is only enough to show that for every vertex cover $C'$ of $\bar{G}$, $V \setminus C'$ is a clique of $G$. Consider an arbitrary pair of nodes $u$ and $v$ (if any) in $V \setminus C'$. Then, since none of $u$ and $v$ is in $C'$, edge $\{u, v\}$ is not in $\bar{G}$. Otherwise $C'$ would have not been a vertex cover of $\bar{G}$. Therefore, edge $\{u, v\}$ is in $G$. This implies that $V \setminus C'$ is a clique of $G$.

(b) Let us assume that we are given an $n$-node graph $G$ such that the minimum vertex cover of $\bar{G}$ is of size $\lceil n/2 \rceil$. Then, the maximum clique of $G$ is of size $\lfloor n/2 \rfloor$. However, the given 2-approximate algorithm might even return set $C := V$ as the vertex cover of $\bar{G}$, without violating the guaranteed approximation ratio. Then, $V \setminus C$ is of size zero while the maximum clique is of size $\Omega(n)$.

## Exercise 2: Weighted Maximum Clique *(10 Points)*

Consider a graph $G = (V, E)$ and a weight function $w : V \to \{1, 2, \ldots, n\}$. Let the weight of a clique $C$ in $G$ be defined as the sum of the weights of the nodes in $C$. Then, the weighted maximum clique problem requires to find a clique with maximum possible weight in $G$.
Let $\mathcal{A}$ be an $\alpha$-approximation algorithm for the (unweighted) maximum clique problem. Using $\mathcal{A}$, provide an $\alpha$-approximation algorithm for the weighted maximum clique problem.

## Sample Solution

We first construct an unweighted graph $G'$ based on $G$. Then, we run the the given algorithm on $G'$ to compute a clique $C'$. Finally having $C'$, we compute a clique $C$ of $G$ with the desired approximation.

Let $V := \{v_1, \ldots, v_n\}$, where node $v_i$ is of weight $w_i$. To construct $G' = (V', E')$, we scan all the nodes in $V$ one by one and transform $G$ to $G'$ as follows. For every $i = 1, \ldots, n$, we replace $v_i$ with a clique $C_i$ of size $w_i$. After replacing all the nodes, for any $i, j \in \{1, \ldots, n\}$, we connect all the nodes of $C_i$ to all the nodes of $C_j$ if and only if $v_i$ and $v_j$ are neighbors in $G$, i.e., $\{v_i, v_j\} \in E$.

Let us assume by running the given algorithm on $G'$, set $C'$ is returned as an $\alpha$-approximate clique of $G'$. Then we construct the final solution $C$ as follows. We add node $v \in V$ into $C$ if and only if at least one of the nodes of the replacing clique of $v$ is in $C'$.

Then, $C$ is shown to be an $\alpha$-approximate solution for $G$ by the following two facts.

1) The weight of $C$ is at least as large as the size of $C'$. This is true because for every node $v_i$ in $C$ we have at most $w_i$ nodes in $C'$.

2) A maximum clique of $G'$ is of size at least the weight of a maximum clique of $G$. It is enough to show that if $G$ has a clique of weight $c$, then $G'$ must have a clique of size $c$. This is immediate from the construction of $G'$ based on $G$.

## Exercise 3: LRU with Potential Function *(12 Points)*

When studying online algorithms, 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, where we are given a fast memory that can hold at most $k$ pages and the goal is to minimize the number of page misses.

We will analyze the competitive ratio of a paging 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 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 sequence of page requests. 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-step $t$ (i.e., after serving requests $\sigma(1), \ldots, \sigma(t)$), let $S_{\text{LRU}}(t)$ be the set of pages in LRU's fast memory and let $S_{\text{OPT}}(t)$ be the set of pages contained in OPT's fast memory. We define $S(t) := S_{\text{LRU}}(t) \setminus S_{\text{OPT}}(t)$.

Further, for each time-step $t$ we assign integer weights $w(p,t) \in \{1, \ldots, k\}$ to each page $p \in S_{\text{LRU}}(t)$ such that for any two pages $p, q \in S_{\text{LRU}}(t), w(p,t) < w(q,t)$ iff the last request for $p$ occurred before the last request for $q$ (i.e., the pages in $S_{\text{LRU}}$ are numbered from $1, \ldots, k$ according to times of their last occurrences, where the least recently used page has weight 1). We define the potential function at time $t$ to be

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

As usual, we define the amortized cost $a_{\text{LRU}}(t)$ for serving request $\sigma(t)$ as

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

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

$$a_{\text{LRU}}(t) \le k \cdot c_{\text{OPT}}(t).$$

## Sample 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 $c_{\mathrm{OPT}} = 0$ and the potential function does not change. If OPT does have a page fault, $c_{\mathrm{OPT}} = 1$ 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 $c_{\mathrm{LRU}} = 0$ and potential function does not change. If LRU has a page fault, $c_{\mathrm{LRU}} = 1$ 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 4: Parallel Merging of Two Sorted Arrays *(2+4+6 Points)*

You are given two sorted arrays $A = [a_1, \ldots, a_n]$ and $B = [b_1, \ldots, b_n]$, each of size $n$. The goal is to merge them into one sorted array $C = [c_1, \ldots, c_{2n}]$ of length $2n$ in the CREW PRAM model.

(a) We first consider the following subproblem. Given an index $i \in \{1, \ldots, n\}$, we want to find the final position $j \in \{1, \ldots, 2n\}$ of the value $a_i$ in the array $C$. Give a fast sequential algorithm to compute $j$. What is the (sequential) running time of your algorithm?

(b) Use the above algorithm to construct a parallel merging algorithm. The work $T_1$ of your algorithm should be at most $O(n \log n)$ and the span (or depth) $T_\infty$ should be (asymptotically) as small as possible. What is the span $T_\infty$ of your algorithm?

(c) We now want to solve the merging problem in constant time (in parallel). Show that by using $O(n)$ processes, the subproblem considered in (a) can be solved in $O(1)$ time. Use this to get a constant-time parallel algorithm to merge the two sorted arrays. How many processors do you need to achieve a constant-time algorithm?

## Sample Solution

Assume that all arrays are sorted in ascending order.

(a) Note that the first $(i-1)$ values in $A$ are before $a_i$ in $C$, because the array $A$ is sorted. Now we have to find out how many values in $B$ are before $a_i$. That is to find the largest index $k$ such that $b_k \leq a_i$. One easy way for this is to compare $a_i$ with the elements of $B$ one by one starting from

$b_1$ and find the index $k$. This will take $O(n)$ time in general, since the size of the array $B$ is $n$. However, we can do it faster using the divide and conquer approach (this is exactly the binary search). We recursively break the array $B$ into two parts of equal size and check in which side $a_i$ falls (and ignore the other side). Using divide and conquer approach we can find the index $k$ in $O(\log n)$ time. Once we find the $k$, then the final position of $a_i$ would be $(i-1)+k+1$ (assuming the array indices starting from 1).

(b) In the above algorithm, we see that one processor can find the final position of a value $a_i$ in $O(\log n)$ time. Now we consider $n$ processors corresponding to each value $a_i$ in $A$ and compute their positions in the output array $C$ in parallel. All the processors can find the final position of every values of $A$ in $O(\log n)$ time. Then in the same way, we compute the position of all the values of $B$ in $C$ using $n$ processors and $O(\log n)$ time. Hence, we can merge the two sorted arrays into one sorted array in $O(\log n)$ time, using $n$ processors. The total work is $T_1 = O(n \log n)$ and the span is $T_\infty = O(\log n)$.

(c) Consider a particular value $a_i$ of $A$ and we want to find the final position of $a_i$ in $C$. Let us take $n$ processors $p_k : k = 1, 2, \ldots, n$. Each processor $p_k$ compares the value $a_i$ with two consecutive values $b_{k-1}$ and $b_k$ in $B$. All the processors do it in parallel. (Note that the array indices starting from 1, so we assume $b_0 = -\infty$ for consistency). Since the values $b_k$ are in ascending order (sorted), there will be only one processor $p_t$ which see that $b_{t-1} \leq a_i$ and $b_t > a_i$. That is there are exactly $t-1$ values in the array $B$ which are smaller than $a_i$. Hence, the processor $p_t$ can decide the final position of $a_i$ which would be $(i-1)+(t-1)+1$ (since there are $i-1$ values smaller than $a_i$ in $A$). The processor $p_t$ can write the value $a_i$ safely in the final array $C$. Note that the processor $p_n$ may observe that $b_n \leq a_i$, then the final position of $a_i$ would be $(i-1)+n+1$. Since all the processors computing this in parallel, it takes constant time. Also we used $n$ processors for this. We can extend this algorithm for all the values in $A$ using $n^2$ processors in $O(1)$ time: for each $a_i$ in $A$, run the algorithm in parallel. For this, we need a total $n^2$ processors and they can write all the values $a_i$ in the correct place in $C$. Notice that there would not be any conflicts when writing in $C$, since a processor $p_t$ only writes the value in one cell of the array $C$. Thus we can put all the values of $A$ in the output array $C$ in constant time.

Now we want to put all the values of $B$ in $C$. Again we can use the same approach as above i.e., we find the right index of a particular value $b_j$ in $B$ by comparing with values in $A$. We have to be a bit careful in this case. During the comparison of a value $b_j$ with two consecutive values $a_{k-1}$ and $a_k$, each processor $p_k$ checks if $a_{k-1} < b_j$ and $a_k \geq b_j$, i.e., processors find index the $t$, for which $a_{t-1} < b_j$ and $a_t \geq b_j$ holds. This "strict" less inequality is necessary to avoid any concurrent writing or conflicts in $C$. The processor $p_t$ which found the index $t$, can decide the final position of $a_i$ as $(j-1)+(t-1)+1$.

Therefore, we can merge the two sorted array of size $n$ into one sorted array in $O(1)$ time using $n^2$ processors.