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

Algorithms Theory
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.

Then, 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 Question (a) to get a vertex cover V'.

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

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 α-approximation algorithm for the (unweighted) maximum clique problem. Using $\mathcal{A}$, provide an α-approximation algorithm for the weighted maximum clique problem.

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_{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 $\sigma(t)$,

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

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_∞ should be (asymptotically) as small as possible. What is the span T_∞ 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?