Approximation Algorithms

Winter term 2021/22

Prof. Dr. Martin Hoefer
Tim Koglin, Lisa Wilhelmi

GOETHE UNIVERSITÄT FRANKFURT AM MAIN
Institute of Computer Science
Algorithms and Complexity

Assignment 4

Issued: 16.11.2021
Due: 23.11.2021, **10:15h**

Exercises marked with * are bonus - they count for your score but not for the sum of points.

Exercise 4.1 *Algorithms for* BinPacking (2 + 3 points)

The algorithm *Best-Fit-Decreasing* for the BinPacking problem operates as follows: For a given instance, the items are sorted in non-increasing order with respect to their sizes $g_i \in [0,1]$. Afterwards, the items are considered consecutively starting with the item of maximum size. Each item is put into the fullest bin in which it fits. If it does not fit in any existing bin, it is put into a new bin.

a) Construct an instance of the BinPacking problem with a minimum number of items for which the *First-Fit-Decreasing* algorithm requires more bins than the *Best-Fit-Decreasing* algorithm. A proof of minimality of the instance is not required.
Hint: The minimal number of items is between 3 and 7.

b) Design an instance such that both the *First-Fit-Decreasing* and the *Best-Fit-Decreasing* algorithm use at least $11/9 \cdot \mathrm{OPT}$ bins.
Hint: It will be sufficient to use item sizes from the set $\{1/2+\varepsilon, 1/4+2\cdot\varepsilon, 1/4+\varepsilon, 1/4-2\cdot\varepsilon\}$, for $\varepsilon > 0$.

Exercise 4.2 *Vector* BinPacking (5 points)

In the *vector* BinPacking problem, n items are given whose sizes are described by d-dimensional vectors $g^i = \left(g^i_1, ..., g^i_d\right)$, where $g^i_j \in [0,1]$ and $d \geq 1$. Each bin has d-dimensional size $(1,1,...,1)$. The goal is to place all n items in as few bins as possible such that for each bin B and each coordinate $j = 1, ..., d$ it holds that $\sum_{i \in B} g^i_j \leq 1$.

Assume that there exists a polynomial-time algorithm with approximation ratio $\alpha > 1$ for the case $d = 1$ (which is equivalent to standard BinPacking as discussed in the lecture). Design a polynomial-time algorithm for general vector BinPacking that uses not more than $\alpha \cdot d \cdot \mathrm{OPT}$ bins.
Hint: For an instance Λ of vector BinPacking *(vBP), think of how to obtain an instance Γ of standard* BinPacking *(BP) such that the (approximated) solution to BP on input Γ is also a feasible solution to vBP on input Λ. Then, consider the largest number of additional bins that must be introduced when transforming a solution to vBP on input Λ to a solution to BP on input Γ.*

Exercise 4.3 IndependentSet *on Trees* (3 + 2 + 3* points)

For an undirected graph $G = (V, E)$, a set $I \subseteq V$ is called an *independent set* if for each pair of vertices $i, j \in I$ it holds that $\{i, j\} \notin E$. In the *maximum-weight* IndependentSet problem, a graph $G = (V, E)$ is given, where each vertex $v \in V$ has a weight $w_v \geq 0$. The goal is to find an independent set $I \subseteq V$ that maximizes the sum of weights of vertices in I.

Suppose that the graph G is a tree. State a dynamic program for maximum-weight IndependentSet of G explicitly.

a) Specify how the maximum total weight of an independent set is determined in a *bottom-up* manner.

b) Specify how the corresponding independent set is constructed in a *top-down* manner.

c)* Now consider the standard IndependentSet problem, i.e., the vertices do not have weights and the goal is to maximize the cardinality of the independent set.

Design a Greedy algorithm for IndependentSet on trees. Prove correctness of the algorithm and determine its running time (a proof of optimality is not required).

Exercise 4.4 *Preparing Exercise Sheets* (3 + 2 points)

Two doctoral students have a pool of n exercises for preparing the weekly exercise sheets, where each exercise $i \in [n]$ has a difficulty $d_i \in \mathbb{N}_{>0}$. Since they want the students to have a steady workload, they want the next two exercise sheets to have the same overall difficulty using all currently available exercises.

a) Design a dynamic program that decides whether or not this is possible.

b) Determine the running time of the dynamic program.