Albert-Ludwigs-Universität, Inst. für Informatik
Prof. Dr. Fabian Kuhn
H. Ghodselahi, Y. Maus

February 09, 2017

Algorithm Theory, Winter Term 2016/17
Problem Set 8

Sample Solution

Exercise 1: Greedy Approximation for Knapsack (17 points)

In the lecture, we have considered the (0-1)-Knapsack problem: There are n items with positive weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack (a bag) of capacity W such that $w_i \leqslant W$ for all $1 \leqslant i \leqslant n$. A feasible solution to the problem is a subset of the items such that their total weight does not exceed W. The objective is to find a feasible solution of maximum possible total value.

Consider the following greedy algorithm:

1. Sort the n items such that $\frac{v_1}{w_1} \geqslant \frac{v_2}{w_2} \geqslant \ldots \geqslant \frac{v_n}{w_n}$.
2. Fill the knapsack sequentially with items in the above sorted order starting with the item with largest value per weight. The algorithm stops either if there are no more items left or it reaches an item $k \leqslant n$ which does not fit, i.e., $w_k > W - \sum_{i=1}^{k-1} w_i$.

a) (5 points) Show that the solution of the greedy algorithm can be arbitrarily bad compared to an optimal solution.

b) (12 points) Using a modification to the greedy algorithm, it is possible to get a 2-approximation for the problem. Present such a modified greedy algorithm and show that it provides approximation factor of 2.

Hint: *For the sake of analyzing your algorithm, you might use the result on fractional knapsack problem (cf. problem set 3, exercise 1, part a).*

Solution

a) Consider two items such that the first one has value $v_1 = 2$ and weight $w_1 = 1$. The second item has value $v_1 = W$ and weight $w_1 = W$. The greedy algorithm picks the first item while the optimal algorithm takes the second item. Therefore, the approximation factor could be arbitrarily bad.

b) Consider the following modified greedy algorithm:

- Sort the n items such that $\frac{v_1}{w_1} \geqslant \frac{v_2}{w_2} \geqslant \ldots \geqslant \frac{v_n}{w_n}$.
- Fill the knapsack sequentially with items in the above sorted order starting with the item with largest value per weight. The algorithm stops either if there are no more items left or it reaches an item $k \leqslant n$ which does not fit, i.e., $w_k > W - \sum_{i=1}^{k-1} w_i$.
- If $v_k \leqslant \sum_{i=1}^{k-1} v_i$ then the algorithm stops. Otherwise, the algorithm removes all the $k-1$ items from the knapsack and put the k-th item in the knapsack and stops.

As you can see, the first and the second steps of the modified algorithm are exactly the same as the greedy algorithm described in the exercise and the last step is only added.

Let I denote an instance of the classic Knapsack problem and I' denote an instance of the fractional Knapsack problem. Further, $S^A(I)$ denote the total value for the modified greedy algorithm. With respect to the modified greedy algorithm we have

$$S^A(I) = \max\left\{v_k, \sum_{i=1}^{k-1} v_i\right\}. \tag{1}$$

Note that $S^A(I) > 0$ since the modified greedy algorithm puts at least one item in the knapsack using the fact that $w_i \leqslant W$ for all $1 \leqslant i \leqslant n$.

Let $S^O(I)$ denote the total value of the optimal solution for the classic version and $S^O(I')$ be the total value of the optimal solution for the fractional version of the Knapsack problem. It is obvious that the total value of the optimal solution for the classic version is upper bounded by the total value of the optimal solution for the fractional version, i.e.,

$$S^O(I) \leqslant S^O(I'). \tag{2}$$

In the Problem Set 3, Exercise 1, we have seen the greedy algorithm for the fractional version of the Knapsack problem that computes an optimal solution. As a recap, that greedy algorithm behaves almost similar to the greedy algorithm described in this exercise for the classic version but the last step. When the algorithm reaches an item $k \leqslant n$ which does not fit, i.e., $w_k > W - \sum_{i=1}^{k-1} w_i$ then it takes as much as possible of the item with the next greatest value per weight. Hence, we have

$$S^O(I') \leqslant \sum_{i=1}^{k} v_i \leqslant 2 \cdot \max\left\{v_k, \sum_{i=1}^{k-1} v_i\right\}. \tag{3}$$

All (1), (2), and (3) together give

$$S^O(I) \leqslant 2 \cdot S^A(I).$$

Exercise 2: LIFO Paging (8 points)

Either give an explanation if the following statement is true or provide acounter example if it is false.

There exists some constant $c \geqslant 1$ such that the Last In First Out (LIFO) paging algorithm is c-competitive.

Solution

It is false! Let us assume that the fast memory (cache) is empty at the beginning and k is the size of the cache. Assume that LIFO is c-competitive for some fixed constant c. We will construct a sequence of requests for which LIFO is not c-competitive. Consider the following input sequence

$$p_1, p_2, \ldots, p_{k-1}, p_k, p_{k+1}, p_k, p_{k+1}, \cdots$$

where $p_i \neq p_j$ for $i \neq j$. As you can see, in the first part there are requests for $k-1$ different pages and then the two pages p_k and p_{k+1} are requested interchangeably until we obtain $(c+1) \cdot (k+1)$ requests in total.

Bounding the Optimal Solution: Consider an algorithm which moves the first k pages $p_1, p_2, \ldots$ $\ldots, p_{k-1}, p_k$ to the cache when they are requested. When the page p_{k+1} is requested for the first time, it leads to a page fault and a page from the cache needs to be evicted and replaced with page p_{k+1}. Evict page p_1 (one could evict any page in $\{p_1, p_2, \ldots, p_{k-1}\}$ since they will not be requested anymore). For the remaining pages of the sequence when they are requested, there are no more page faults since the pages p_k and p_{k+1} are already in the cache. Consequently, the number of page faults of this algorithm is only $k+1$. Thus we know that the optimal solution needs at most $k+1$ page faults.

LIFO algorithm The LIFO algorithm similarly moves all pages $p_1, p_2, \ldots, p_{k-1}, p_k$ to the cache when they are requested. From the time p_{k+1} is requested for the first time and onwards, all remaining requests each leads to a page fault because LIFO just evicted the requested page in the step before. Thus, the number of page faults by LIFO is $(c+1) \cdot (k+1)$. Therefore the LIFO algorithm is at least by a factor $c+1$ worse than the optimal solution on the given input sequence, i.e., it is not c-competitive.

Exercise 3: Online Vertex Cover (15+5* points)

Given a graph $G = (V, E)$. A set $S \subseteq V$ is called a *vertex cover* if and only if for every edge $\{u, v\} \in E$ at least one of its endpoints is in S. The minimum vertex cover problem is to find such a set S of minimum size.

We are considering the following online version of the minimum vertex cover problem. Initially, we are given the set of nodes V and an empty vertex cover $S = \varnothing$. Then, the edges appear one-by-one in an online fashion. When a new edge $\{u, v\}$ appears, the algorithm needs to guarantee that the edge is covered (i.e., if this is not already the case, at least one of the two nodes u and v needs to be added to S). Once a node is in S it cannot be removed from S.

a) (15 points) Provide an online algorithm with competitive ratio at most 2. That is, your online algorithm needs to guarantee at all times that the vertex cover S is at most by a factor 2 larger than a current optimal vertex cover. Discuss the correctness of your algorithm.

b) (5 bonus points) Show that there does not exist any online algorithm that can provide a better competitive ratio than 2.

Solution

a) Consider the following online algorithm: consider a new edge that appears. Our online algorithm checks whether the new edge is already covered, i.e., it has at least one endpoint in S or not (note that S is empty at the beginning). In the case that the edge is not covered then the algorithm adds **both** endpoints of the new edge to S; if the edge is already covered then the algorithm adds no point to S.

Correctness: Our online algorithm guarantees that any edge that appears will have at least one endpoint in S. Therefore, the set S is a vertex cover of the graph.

Claim: The above online algorithm provides a competitive ratio of 2.

Proof. Consider a set M of edges that is empty at the beginning. We determine what edges are in M regarding to the online algorithm. If our online algorithm puts both endpoints of a new edge in S then the edge is put in M. Hence we have

$$|M| = \frac{|S|}{2}.$$

We have seen in the lecture that the size of a matching of a graph is upper bounded by the size of a vertex cover of the graph (see slide no. 6 of Graph Algorithms Part 5). Hence the claim holds if we show that M is a matching. Our online algorithm guarantees that for any two edges $\{u, v\}$ and $\{u', v'\}$ in M, the edges are not incident. More precisely, assume first $\{u, v\}$ appears and its both endpoints are added to S. Therefore the edge is added to M. Later the edge $\{u', v'\}$ appears. It will be added to M only if $u \neq u'$ and $v \neq v'$ w.r.t. the online algorithm. This guarantees that these two edges are not incident and thus M is a matching. □

b) Let us fix OPT to be an optimal offline algorithm and ALG to be any online algorithm. Consider two distinct edges that are incident. We (as an adversary) construct a scenario where ALG has to put two vertices in S while OPT can cover both edges by only one vertex. We send the first edge. If ALG puts both endpoints of the first edge in S then the second edge can be connected to any endpoint of the first edge. But, if ALG chooses one of the endpoint of the first edge to be in S then we connect the second edge to the endpoint of the first edge that is not in S. So, the second

edge has already no endpoint in S and ALG has to put at least one of the endpoint of the second edge in S. Therefore ALG returns S whose size is at least 2.

By contrast, OPT chooses the middle vertex to be in the vertex cover set and it is enough to cover both edges. Note that OPT can do this since it is offline and knows the input sequence in advance. Therefore, the lower bound for online vertex cover is 2.