Albert-Ludwigs-Universität, Inst. für Informatik
Prof. Dr. Fabian Kuhn
M. Ahmadi, O. Saukh, A. R. Molla November 23, 2015

Algorithm Theory, Winter Term 2015/16 Problem Set 4 - Sample Solution

Exercise 1: Filling Two Knapsacks (6 points)

In this problem, we consider a variation of the knapsack problem, where we have two instead of only one knapsack. Formally, we have items $1, \dots, n$ and each item i has a positive integer *weight* $w_i \in \mathbb{N}$ and a positive *value* $v_i > 0$. Further, we have two knapsacks of capacities W_1 and W_2. We need to pack the items into the knapsacks such that

- Each item is in at most one of the knapsacks
- For $j \in \{1, 2\}$, the *total weight* of the items in knapsack j is at most W_j.
- The *total value* of the items that are packed in either knapsack is maximized.

(a) (2 points) When first looking at the problem, one could think that it is equivalent to the standard knapsack problem with one knapsack of capacity $W' := W_1 + W_2$. Prove that this is not true by showing that in some cases, the total value that can be packed into one knapsack of capacity $W' = W_1 + W_2$ can be *arbitrarily* larger than the total value that can be packed into two knapsacks of capacities W_1 and W_2.

(b) (4 points) Give a dynamic programming algorithm that *optimally* solves the problem (for integer weights). What is the running time of your algorithm?

Solution

a) Consider three items available for filling knapsack: $(W_1, 1)$, $(W_2, 1)$ and $(W_1 + W_2, k)$, where $k > 2$ is arbitrary large. If we consider two knapsacks with capacities W_1 and W_2 and these three items, the optimal solution for this problem instance is to pack first element into first knapsack and the second into the second one, what gives the total value of 2. It is easy to see that the third item of weight $W_1 + W_2$ can not be packed into knapsacks of capacity W_1 or W_2 individually, but to the knapsack of capacity $W_1 + W_2$. Moreover, we can choose k arbitrarily large. Therefore, the total value that can be packed into one knapsack of capacity $W_1 + W_2$ is arbitrarily larger than the total value (i.e., 2) that can be packed into two knapsacks of capacities W_1 and W_2.

b) The solution is similar to the knapsack problem in the lecture notes, where optimal solution is obtained from the optimal solutions of the sub-problems. Only, instead of one knapsack, now we have two knapsacks. Let $\mathrm{OPT}(k, x, y)$ denote optimal value (solution) achievable with items $1, \dots, k$ and two knapsacks of capacities x and y. We define

$$\forall k \in \{0, 1, \dots, n\} : \quad OPT(k, 0, 0) = 0$$

and

$$\forall x \in \{0, 1, \dots, W_1\}, \forall y \in \{0, 1, \dots, W_2\} : \quad OPT(0, x, y) = 0.$$

Assume now, that all optimal solutions for all capacities up to x and y, respectively, for the first and second knapsack and also for the first $k-1$ items are known.

Let us now consider the k^{th} item. It can either be ignored (not get picked into any of the knapsacks), or it is picked into one of the knapsacks. For each of these cases one of the following holds:

(a) If k^{th} item is not picked, then

$$OPT(k, x, y) = OPT(k-1, x, y).$$

(b) If the item is picked into the first knapsack, then

$$OPT(k, x, y) = OPT(k-1, x - w_k, y) + v_k.$$

(c) If the item is picked into the second knapsack, then

$$OPT(k, x, y) = OPT(k-1, x, y - w_k) + v_k.$$

Since we are looking for an optimal solution – the largest possible value inside of both knapsacks – we get a recursive formula for our problem:

$$\begin{aligned} OPT(k, x, y) = \max \big\{ & OPT(k-1, x, y), \\ & OPT(k-1, x - w_k, y) + v_k, \\ & OPT(k-1, x, y - w_k) + v_k \big\} \end{aligned}$$

In order to solve this problem with a dynamic programming approach, it is proposed to fill out a 3D table of size $n \times W_1 \times W_2$ and fill it using the recurrence relation above row-by-row. The answer to our problem will be in the last cell of this table.

Running Time: It is easy to see that the table which needs to be filled is a three dimensional matrix with size $O(n \cdot W_1 \cdot W_2)$. Since the time for calculating each table entry is constant, the total running time is $O(nW_1W_2)$.

Exercise 2: Game Strategy (6 points)

Consider the following two-player game: There is a row of n objects (assuming n is even) with values $v_1, v_2, \ldots, v_n$. In the game, the two players make moves alternatively. In each odd move, the first player either selects the first or the last object of the row and removes it permanently from the row. In even turns, the opponent plays the game in the same way. Each time the player picks some object, it receives its value.

a) (4 points) Devise a dynamic programming algorithm to determine the maximum value that the first player (i.e., the player starting the game) can receive by the end of the game.

b) (2 points) What is the running time for the algorithm?

Solution

a) Let us define $P(i, j)$ to be the maximum value that a player (any player!) can get by playing on successive objects starting from object i and ending with object j, inclusively, by making the first move. Suppose the first player chooses object i with value v_i, and the remaining objects in the row are objects $i+1$ to j. The opponent either chooses object $i+1$ or object j. Note that the opponent is smart enough and always chooses the object which yields the minimum value for the first player. If the opponent takes object $i+1$, the remaining objects are objects $i+2$ to j, on which the first player's maximum value is denoted by $P(i+2, j)$. On the other hand, if the opponent takes object

j, the maximum is $P(i+1, j-1)$. Therefore, the maximum amount first player can get when he chooses object i is

$$v_i + \min\{P(i+2,\, j),\, P(i+1,\, j-1)\}.$$

Similarly, the maximum amount first player can get if chooses object j is

$$v_j + \min\{P(i+1,\, j-1),\, P(i,\, j-2)\}.$$

Concluding from the above statements, the recurrence relation for an optimal solution is as follows.

$$P(i,j) = \max\big\{v_i + \min\{P(i+2,\, j),\, P(i+1,\, j-1)\},\, v_j + \min\{P(i+1,\, j-1),\, P(i,\, j-2)\}\big\}$$

And the base cases are

$$P(i,i) = v_i \quad \text{and} \quad P(i,i+1) = \max\{v_i,\, v_{i+1}\}.$$

Now, let us use this formula for dynamic programming approach, since it can help us avoid computing the same optimal solutions for the subproblems multiple times. To solve the problem, we need to fill a 2D table of size $n \times n$, where entry $[i,j]$ in the table will contain the optimal solution $OPT(i,j)$.

First, we initialize our table with zeros and fill it afterwords using the following strategy. On each iteration k of filling the table, we compute $P(i,j)$ using the aforementioned recurrence relation for all i and j, such that $i - j = k$ and $i < j$. Our result in table is in place $[1, n]$.

b) From design of this algorithm, we can see the only thing that has to be done is filling out the table of size $n \times n$. Calculating each table entry takes a constant time, therefore, filling out the whole table takes $O(n^2)$.