Journal of Combinatorial Optimization, 8, 5–11, 2004
© 2004 Kluwer Academic Publishers. Manufactured in The Netherlands.

Improved Dynamic Programming in Connection with an FPTAS for the Knapsack Problem

HANS KELLERER hans.kellerer@uni-graz.at
ULRICH PFERSCHY pferschy@uni-graz.at
Department of Statistics and Operations Research,University of Graz, Universitätsstr. 15, A-8010 Graz, Austria

Received January 18, 2000; Revised October 22, 2001; Accepted January 7, 2002

Abstract. A vector merging problem is introduced where two vectors of length n are merged such that the k-th entry of the new vector is the minimum over ℓ of the ℓ-th entry of the first vector plus the sum of the first $k - \ell + 1$ entries of the second vector. For this problem a new algorithm with $O(n \log n)$ running time is presented thus improving upon the straightforward $O(n^2)$ time bound.

The vector merging problem can appear in different settings of dynamic programming. In particular, it is applied for a recent fully polynomial time approximation scheme (FPTAS) for the classical 0–1 knapsack problem by the same authors.

Keywords: dynamic programming, knapsack problem, fully polynomial approximation scheme

1. Introduction

The classical *0–1 knapsack problem* (KP) is defined by

$$\begin{aligned} \text{(KP) maximize} \quad & \sum_{i=1}^{n} p_i x_i \\ \text{subject to} \quad & \sum_{i=1}^{n} w_i x_i \le c \\ & x_i \in \{0, 1\}, \quad i = 1, \dots, n, \end{aligned} \tag{1}$$

with *profits* p_i, *weights* w_i and *capacity* c being positive integers. It can be interpreted as filling a knapsack with a subset of the item set $\{1, \dots, n\}$ maximizing the *profit* in the knapsack such that its *weight* is not greater than the *capacity* c.

Recently, we presented in Kellerer and Pferschy (1999) a new fully polynomial time approximation scheme (*FPTAS*) for this $\mathcal{NP}$–hard problem considerably reducing the space requirement. Also the running time was reduced under very reasonable assumptions.

In this note we present an efficient algorithm for a certain problem of merging vectors in a dynamic programming context. In particular, the problem occurs in the dynamic programming procedure of the *FPTAS* in Kellerer and Pferschy (1999). The application of the new method in the algorithmic framework of Kellerer and Pferschy (1999) also corrects an erroneous detail of the previously used procedure. Moreover, the running time is rectified

Table 1. Complexity of *FPTAS* for (KP).

Author	Running time	Space
Lawler (1979)	$O(n \log(1/\varepsilon) + 1/\varepsilon^4)$	$O(n + 1/\varepsilon^3)$
Magazine and Oguz (1981)	$O(n^2 \log n \cdot 1/\varepsilon)$	$O(n \cdot 1/\varepsilon)$
Kellerer and Pferschy (1999)	$O(n \min\{\log n, \log(1/\varepsilon)\} + 1/\varepsilon^2 \min\{n, 1/\varepsilon^2\})$	$O(n + 1/\varepsilon^2)$
This note	$O(n \min\{\log n, \log(1/\varepsilon)\} + 1/\varepsilon^2 \log(1/\varepsilon)$ $\min\{n, 1/\varepsilon \log(1/\varepsilon)\})$	$O(n + 1/\varepsilon^2)$

adding a factor of $\log(1/\varepsilon)$ to the value given in Kellerer and Pferschy (1999). The actual running time of the algorithm in Kellerer and Pferschy (1999) should have been $O(1/\varepsilon^4)$.

Table 1 summarizes the time and space complexity of previous and the new approximation scheme for (KP). A recent review of optimal solution methods for (KP) is given in Martello et al. (2000).

It can be seen that our method is clearly superior to the one in Lawler (1979). Compared to Magazine and Oguz (1981) we get an improvement in the crucial aspect of space for $n \geq 1/\varepsilon$. In this case also the running time is improved, which means that the necessary addition of the $\log(1/\varepsilon)$ factor does not change the competitiveness towards previous schemes. As noted before, the assumption that $n \geq 1/\varepsilon$ is rather practical because for a problem with a moderate number of items and a very high accuracy optimal solution methods could be successfully applied.

After defining the vector merging problem below we will introduce the new algorithm through a sequence of three algorithms with increasing structural complexity in Section 2. The final algorithm will solve the problem in $O(n \log n)$ time and thus reduce significantly the straightforward $O(n^2)$ bound.

In Section 3 it will be shown how the vector merging problem and the new algorithm can be used to improve the dynamic programming of the *FPTAS* for (KP).

Problem VectorMerge VM: Given two vectors A from $A[1]$ to $A[n]$ and B from $B[0]$ to $B[n-1]$ with $0 \leq B[0] \leq B[1] \leq \cdots \leq B[n-1]$ we would like to compute a vector C of the same size defined by

$$\begin{aligned}
C[1] &:= A[1] + B[0] \\
C[2] &:= \min\{A[1] + B[0] + B[1],\ A[2] + B[0]\} \\
&\ \vdots \\
C[n] &:= \min\{A[1] + B[0] + B[1] + \cdots + B[n-1], \ldots, \\
&\qquad A[n-1] + B[0] + B[1], A[n] + B[0]\}.
\end{aligned}$$

Equivalently, we can write

$$C[k] := \min\left\{ A[\ell] + \sum_{j=0}^{k-\ell} B[j] \;\middle|\; \ell = 1, \ldots, k \right\} \quad \text{for } k = 1, \ldots, n.$$

2. Improved dynamic programming for VM

The computation of vector C can be done trivially in $O(n^2)$ time, e.g. by evaluating the minimum over at most n values explicitly for each entry $C[k]$ and storing the required partial sums for the next iteration with $C[k+1]$. In this section we will give an algorithm with an improved time complexity of $O(n \log n)$.

To provide a better intuition of the applied technique we will start with a brief description of a straightforward solution method still requiring $O(n^2)$ time but performing the computation in a different order. In a first iteration we check for all entries of C whether $A[1]$ plus the appropriate entries of B yield new minima, then we do the same for $A[2]$ and so on. Basically, we compute the values in the above definition columnwise instead of rowwise. For notational convenience we introduce a vector *origin* indicating finally that the minimum defining $C[k]$ is attained by $A[\mathit{origin}[k]]$ plus the appropriate entries of B. Note that C is completely determined by *origin*.

Representing C by vector *origin* we can identify a monotonicity property in problem VM which will be exploited in the improved algorithm.

Observation 1. There exists a solution of VM with

$$\mathit{origin}[j] \le \mathit{origin}[j+1] \quad \text{for } \ j = 1, \dots, n-1.$$

Proof: Assume otherwise that in every solution there is some j with $i := \mathit{origin}[j] > \mathit{origin}[j+1] =: i'$. Then we would have

$$\begin{aligned} C[j+1] &= A[i'] + B[0] + B[1] + \cdots + B[j+1-i'] \\ &\le A[i] + B[0] + B[1] + \cdots + B[j+1-i]. \end{aligned}$$

Considering that $B[j+1-i'] \ge B[j+1-i]$ the inequality holds even more so after subtracting the last term from both sums. This yields

$$\begin{aligned} A[i'] + B[0] + B[1] + \cdots + B[j-i'] \le A[i] + B[0] + B[1] \\ + \cdots + B[j-i] = C[j], \end{aligned}$$

which is either a contradiction to the definition of C, or (in the case of equality) there also exists a solution where this violation of the desired property can be avoided by a different breaking of ties in the minimum. □

The following algorithms will determine a solution fulfilling the property of Observation 1. In the application of VM to the *FPTAS* for the knapsack problem in Section 3, but very likely also in other applications of this dynamic programming structure, it will turn out that the vector *origin* contains identical values for a large number of consecutive entries before increasing again. This gives rise to the idea not to store and compute each

```
    for $j := 1$ to $n$ do
        $C[j] := A[1] + B[0] + B[1] + \cdots + B[j-1]$
    $a[1] := 1, b[1] := n, pred[1] := 0$
    $last := 1$        *last denotes the index of the most recent non-empty interval*
    for $i := 2$ to $n$ do
        if $C[n] > A[i] + B[0] + B[1] + \cdots + B[n-i]$ then
            $b[i] := n$
            $j := n - 1$
            while $C[j] > A[i] + B[0] + B[1] + \cdots + B[j-i]$ and $j \geq i$ do
                $C[j] := A[i] + B[0] + B[1] + \cdots + B[j-i]$
                if $j = a[last]$ then
                    $a[last] := \infty, b[last] := \infty$
                    $last := pred[last]$
                $j := j - 1$
            end while
            $b[last] := j$
            $a[i] := j + 1, pred[i] := last$
            $last := i$
        else
            $a[i] := \infty, b[i] := \infty$
    end for
*reconstruction of origin:*
    while $last > 0$ do
        for $j := b[last]$ down to $a[last]$ do
            $origin[j] := last$
        $last := pred[last]$
    end while
```

Figure 1. Algorithm **VM-interval**.

of the values of *origin* explicitly but to try to find only the endpoints of intervals where *origin* remains constant. This means that for each i we define the endpoints of an interval $a[i], b[i]$ such that for every j with $a[i] \leq j \leq b[i]$ there is $origin[j] = i$ and hence $C[j] = A[i] + B[0] + B[1] + \cdots + B[j-i]$.

Clearly, some (more likely many) values of i will not appear in *origin* at all and hence be assigned an empty interval with $a[i] = b[i] = \infty$. To achieve an efficient handling of the non-empty intervals we will link them together by defining a vector *pred* where an entry $pred[i] = s$ means that $a[i], b[i] \neq \infty$, $a[k] = b[k] = \infty$ for $s < k < i$ and $a[s], b[s] \neq \infty$. For $a[i] \neq \infty$ there always must be $a[i] = b[pred[i]] + 1$.

The solution of VM by a computation of these intervals is performed by algorithm **VM-interval** which is described in detail in figure 1. It is easy to see that also this interval based algorithm still requires $O(n^2)$ running time. However, we can improve the computation in the following way.

Note that if $C[j] > A[i] + B[0] + B[1] + \cdots + B[j-i]$ holds for $j = a[k]$ for some k, it follows from Observation 1 that this inequality also holds for all $j > a[k]$. Hence, it is sufficient to check this condition for all starting points $a[k]$ of non-empty intervals. As soon as we find a position $a[k]$ where the condition is violated we have to find the exact new values of $b[k]$ and $a[i]$. This can be done by binary search between $a[k]$ and the old value

```
for $j := 1$ to $n$ do
  $C[j] := A[1] + B[0] + B[1] + \cdots + B[j-1]$
$a[1] := 1, b[1] := n, pred[1] := 0$
$last := 1$        *last denotes the index of the most recent non-empty interval*
for $i := 2$ to $n$ do
  if $C[n] > A[i] + B[0] + B[1] + \cdots + B[n-i]$ then
      $b[i] := n$
      $j := a[last]$
      while $C[j] > A[i] + B[0] + B[1] + \cdots + B[j-i]$ and $j \geq i$ do
          $C[j] := A[i] + B[0] + B[1] + \cdots + B[j-i]$
          $a[last] := \infty, b[last] := \infty$
          $last := pred[last]$
          $j := a[last]$
      end while
      if $j \geq i$ then
          Perform binary search to find the largest value $j \in a[last], \ldots, b[last]$
              with $C[j] \leq A[i] + B[0] + B[1] + \cdots + B[j-i]$
          $b[last] := j, a[i] := j + 1$
      else
          $b[last] := i - 1, a[i] := i$
      $pred[i] := last$
      $last := i$
  else
      $a[i] := \infty, b[i] := \infty$
*reconstruction of origin:* as in **VM-interval**
```

Figure 2. Algorithm **VM-improved**.

of $b[k]$. The resulting algorithm **VM-improved** is presented in figure 2. The reconstruction of *origin* is omitted since it is exactly the same as in **VM-interval**.

To analyze the running time of **VM-improved** we consider at first the comparisons of $C[j]$ in the while-loop for some previously detected interval endpoints. If the inequality is true, then the corresponding interval is deleted and never considered again. If the inequality is false, then the loop is stopped. Hence, the total number of these comparisons is linear in n because there can be at most n positive results (one for every interval) and at most one negative result in each of the n iterations of the for-loop.

Thus the running time of **VM-improved** is dominated by the at most n executions of the binary search over $O(n)$ values. In order to perform the binary search procedure in $O(\log n)$ time, i.e. to answer every query of the binary search in constant time, we generate an n-dimensional auxiliary array where every entry j, $j = 0, 1, \ldots, n-1$, contains the value $B[0] + B[1] + \cdots + B[j]$. Summarizing, we have shown

Theorem 2. *Algorithm* **VM-improved** *can be performed in* $O(n \log n)$ *time.*

3. Improved dynamic programming for the *FPTAS* of (KP)

It is beyond the scope of this paper to completely repeat the complicated structure of the *FPTAS* presented in Kellerer and Pferschy (1999). However, we will briefly sketch the dynamic programming part and the data of the approximation algorithm as far as necessary to understand the relation to problem VM.

The central routine of the *FPTAS* is a procedure performing dynamic programming by profits. This means that after computing an upper bound *UB* on the optimal solution value of (KP) a dynamic programming function $z_j(p)$ is introduced for $j = 1, \dots, n$ and $p = 0, 1, \dots, UB$ such that $z_j(p) = w$ indicates that there exists a subset of items from $\{1, \dots, j\}$ with total profit p and total weight w. Furthermore, w is the minimal weight among all such subsets. It is well known that z can be computed by the recursion

$$z_{j+1}(p) := \begin{cases} z_j(p) & \text{for } p < p_{j+1} \\ \min\{z_j(p), z_j(p - p_{j+1}) + w_{j+1}\} & \text{for } p \geq p_{j+1} \end{cases} \tag{2}$$

after initializing $z_0(0) := 0$ and $z_0(p) := c + 1$ for $p = 1, \dots, UB$. The optimal solution value is given by $\max\{p \mid z_n(p) \leq c\}$.

For a given accuracy ε the set of items is partitioned such that $O(1/\varepsilon^2)$ items remain to be considered. Moreover, after scaling there remain only $O(1/\varepsilon \log(1/\varepsilon))$ different profit values for these items. For each of these profit values a set of items with identical profits and different weights is given.

Performing the dynamic programming recursion (2) for this modified item set can be done in a more efficient way. Clearly, the computation of $z_{j+1}(p)$ can be done in an arbitrary order of the profit values p. It will be convenient to evaluate in one iteration those with the same residual value of the division of p by the profit p_t of the item currently added to the problem. This means that for every residual value $r = 0, 1, \dots, p_t - 1$ we consider the new entries $z_{j+1}(r), z_{j+1}(r + p_t), z_{j+1}(r + 2p_t), \dots$ in one run.

Furthermore, we will consider all items with identical profit p_t and weights $w_1^t \leq w_2^t \leq w_m^t$ together in one iteration. Let the corresponding items be denoted by $j + 1, j + 2, \dots, j + m$. For simplicity, we will illustrate the case $r = 0$. In the resulting iteration this means e.g. that instead of computing

$$\begin{aligned} z_{j+1}(p_t) &:= \min\{z_j(p_t), z_j(0) + w_1^t\}, \\ z_{j+1}(2p_t) &:= \min\{z_j(2p_t), z_j(p_t) + w_1^t\} \quad \text{and} \\ z_{j+2}(2p_t) &:= \min\{z_{j+1}(2p_t), z_{j+1}(p_t) + w_2^t\}, \end{aligned}$$

we determine

$$z_{j+2}(2p_t) := \min\{z_j(2p_t), z_j(p_t) + w_1^t, z_j(0) + w_1^t + w_2^t\}.$$

This formulation of the recursion has exactly the same structure as the vector merging problem VM. Let $C[k] := z_{j+m}((k-1)p_t)$ and $A[k] := z_j((k-1)p_t)$ for $k = 1, \dots, \lfloor UB/p_t \rfloor + 1$. Vector B is chosen as $B[0] := 0$, $B[k] := w_k^t$ for $k = 1, \dots, m$ and $B[k] := \infty$ for $k > m$. The analogous treatment of the cases $r > 0$ is straightforward.

It follows from Theorem 2 that every iteration for one profit value p_t and one residual value r can be performed in $O(UB/p_t \cdot \log(UB/p_t))$ time. Iterating over all values of r yields a running time of $O(UB \cdot \log(UB))$ for each of the $O(\min\{n, 1/\varepsilon \log(1/\varepsilon)\})$ different profit values.

The scaling of the profit values guarantees that every profit value is a multiple of $UB\varepsilon^2$. Hence, the computation of z can be reduced to multiples of $UB\varepsilon^2$ instead of considering all integer values. Clearly, the dynamic programming function for this scaled instance has $O(1/\varepsilon^2)$ entries. Summarizing, the running time complexity of the *FPTAS* is given by $O(1/\varepsilon^3 \log^2(1/\varepsilon))$.

This complexity would be sufficient to reach the desired running time for the computation of an approximate solution value. However, the *FPTAS* in Kellerer and Pferschy (1999) uses a dynamic programming procedure, which is called many times within a recursive scheme with different parameters, to construct also the corresponding solution set. The requirements for this procedure to be used in the algorithmic framework and notation of Kellerer and Pferschy (1999) are the following.

Consider a knapsack problem with item set L' containing $D(L')$ different profit values which are all multiples of $UB\varepsilon^2$. Determine the optimal solution value of this problem, where an upper profit bound P' is given, in $O(D(L') \cdot P'/(UB\varepsilon^2) \cdot \log(P'/(UB\varepsilon)))$ time. It is easy to see that the running time of the above construction applying the appropriate instance of VM for the restricted knapsack problem yields exactly this bound.

Acknowledgments

We would like to thank two anonymous referees who helped a lot to improve the presentation of the paper.

References

H. Kellerer and U. Pferschy, "A new fully polynomial time approximation scheme for the knapsack problem," *Journal of Combinatorial Optimization*, vol. 3, pp. 59–71, 1999.

E. Lawler, "Fast approximation algorithms for knapsack problems," *Mathematics of Operations Research*, vol. 4, pp. 339–356, 1979.

M.J. Magazine and O. Oguz, "A fully polynomial approximation algorithm for the 0–1 knapsack problem," *European Journal of Operational Research*, vol. 8, pp. 270–273, 1981.

S. Martello, D. Pisinger, and P. Toth, "New trends in exact algorithms for the 0–1 knapsack problem," *European Journal of Operational Research*, vol. 123, pp. 325–332, 2000.