Computers & Industrial Engineering xxx (2010) xxx–xxx

Contents lists available at ScienceDirect

Computers & Industrial Engineering

journal homepage: www.elsevier.com/locate/caie

A dynamic programming method with lists for the knapsack sharing problem

V. Boyer *, D. El Baz, M. Elkihel

CNRS, LAAS, 7 avenue du Colonel Roche, F-31077 Toulouse, France
Université de Toulouse, UPS, INSA, INP, ISAE, LAAS, F-31077 Toulouse, France

ARTICLE INFO

Article history:
Available online xxxx

Keywords:
Knapsack sharing problem
Combinatorial optimization
Max–min programming
Dynamic programming

ABSTRACT

In this paper, we propose a method to solve exactly the knapsack sharing problem (KSP) by using dynamic programming. The original problem (*KSP*) is decomposed into a set of knapsack problems. Our method is tested on correlated and uncorrelated instances from the literature. Computational results show that our method is able to find an optimal solution of large instances within reasonable computing time and low memory occupancy.

1. Introduction

The knapsack sharing problem (KSP) is a max–min mathematical programming problem with a knapsack constraint (see Brown, 1979, 1991, 1994). The KSP is NP-complete. The KSP occurs when resources have to be shared or distributed fairly to several entities, e.g. distribution of scarce resources like ammunition or gasoline among different military units (see Brown, 1979) or budget shared between districts of a city (see Yamada, Futakawa, & Kataoka, 1997, 1998). The KSP is composed of n items divided into m different classes. Each class $\mathcal{N}_i$ has a cardinality n_i with $\sum_{i\in\mathcal{M}} n_i = n$ and $\mathcal{M} = \{1, 2, \ldots, m\}$. Each item $j \in \mathcal{N}_i$ is associated with:

- a profit p_{ij},
- a weight w_{ij},
- a decision variable $x_{ij} \in \{0, 1\}$.

We wish to determine a subset of items to be included in the knapsack of capacity C, so that the minimum profit associated with the different class is maximised. The KSP can be formulated as follows:

$$(KSP)\begin{cases} \max \min\limits_{i\in\mathcal{M}} \left\{ \sum\limits_{j\in\mathcal{N}_i} p_{ij}\cdot x_{ij} \right\} = z(KSP), \\ s.t. \sum\limits_{i\in\mathcal{M}} \sum\limits_{j\in\mathcal{N}_i} w_{ij}\cdot x_{ij} \leqslant C, \\ x_{ij} \in \{0,1\} \text{ for } i \in \mathcal{M} \text{ and } j \in \mathcal{N}_i, \end{cases} \tag{1.1}$$

where $z(KSP)$ denotes the optimal value of the problem (*KSP*) and for $i \in \mathcal{M}$ and $j \in \mathcal{N}_i$, w_{ij}, p_{ij} and C are positive integers. Furthermore, we assume that $\sum_{i\in\mathcal{M}}\sum_{j\in\mathcal{N}_i} w_{ij} > C$ and $\max_{i\in\mathcal{M}, j\in\mathcal{N}_i}\{w_{ij}\} \leqslant C$.

A common way to solve the KSP consists of its decomposition into knapsack problems (see for examples Hifi & Sadfi, 2002; Yamada et al., 1998). Indeed, for a class $i \in \mathcal{M}$, we define the following problem:

$$(KP_i(C_i))\begin{cases} \max \sum\limits_{j\in\mathcal{N}_i} p_{ij}\cdot x_{ij} = z(KP_i(C_i)), \\ s.t. \sum\limits_{j\in\mathcal{N}_i} w_{ij}\cdot x_{ij} \leqslant C_i, \\ x_{ij} \in \{0,1\} \quad j \in \mathcal{N}_i. \end{cases} \tag{1.2}$$

The objective is then to find $(C_1^*, C_2^*, \ldots, C_m^*)$ such that

$$\sum_{i\in\mathcal{M}} C_i^* \leqslant C,$$

and

$$\min_{i\in\mathcal{M}}\{z(KP_i(C_i^*))\} = z(KSP),$$

where for a problem $\mathcal{P}, z(\mathcal{P})$ represents its optimal value. An upper bound and a lower bound of $z(\mathcal{P})$ will be denoted, respectively, by $\bar{z}(\mathcal{P})$ and $\underline{z}(\mathcal{P})$, respectively.

Hifi et al. have proposed to solve the knapsack problems $(KP_i(C_i))_{i\in\mathcal{M}}$ via a dense dynamic programming algorithm in Hifi and Sadfi (2002) and Hifi et al. (2005). Their method starts with $C_i = 0, i \in \mathcal{M}$, and increases regularly the capacities until $\sum_{i\in\mathcal{M}} C_i > C$.

In this article, we propose an algorithm for solving the KSP. Our algorithm is based on a dynamic programming procedure with dominance technique to solve the knapsack problems $(KP_i(C_i))_{i\in\mathcal{M}}$. Our algorithm starts by solving the problems $(KP_i(C_i))_{i\in\mathcal{M}}$ with $C_i \geqslant C_i^*, i \in \mathcal{M}$ and $\sum_{i\in\mathcal{M}} C_i \geqslant C$. At each step, we try to decrease

* Corresponding author. Tel.: +33 6 11 23 24 75.
E-mail addresses: vboyer@laas.fr (V. Boyer), elbaz@laas.fr (D. El Baz), elkihel@laas.fr (M. Elkihel).

0360-8352/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cie.2010.10.015

the values of $(C_i)_{i\in\mathcal{M}}$, towards $(C_i^*)_{i\in\mathcal{M}}$. The use of lists permits one to reduce the memory occupancy; the expected benefit being the solution of large instances.

Without loss of generality, we consider in the sequel that the items in a class $i \in \mathcal{M}$ are sorted according to decreasing ratios $\frac{p_{ij}}{w_{ij}}, j \in \mathcal{N}_i$.

In Section 2, we present the dynamic programming algorithm used to solve the problems $(KP_i)_{i\in\mathcal{M}}$. Section 3 deals with the algorithm we propose for the KSP. Computational results are displayed and analyzed in Section 4. Some conclusions and perspectives are presented in Section 5.

2. Basic dynamic programming procedure

In order to solve the problems $(KP_i(C_i))_{i\in\mathcal{M}}$, we use a dynamic programming algorithm with dominance (see Bellman, 1957; Boyer, El Baz, & Elkihel, 2009, 2010; El Baz & Elkihel, 2005).

2.1. Lists construction

We recall that $i \in \mathcal{M}$ denotes the index of the ith class of (KSP).

A list $\mathcal{L}_{ik}$ is associated with each step $k \in \mathcal{N}_i$:

$$\mathcal{L}_{ik} = \left\{ (w,p) | w = \sum_{j=1}^{k} w_{ij}\cdot x_{ij} \leqslant C_i \text{ and } p = \sum_{j=1}^{k} p_{ij}\cdot x_{ij}, x_{ij} \in \{0,1\},\ j \in \{1,2,\ldots,k\} \right\}. \tag{2.1}$$

The algorithm begins with the lists $\mathcal{L}_{i0} = \{(0,0)\}$.

At each step $k \in \mathcal{N}_i$, the new list $\mathcal{L}_{ik}$ is obtained as follows:

$$\mathcal{L}_{ik} = \mathcal{L}_{i(k-1)} \cup \{(w+w_{ik}, p+p_{ik}) | (w,p) \in \mathcal{L}_{i(k-1)}, w + w_{ik} \leqslant C_i\}.$$

Notation. For simplicity of presentation the above equation will also be written as follows in the sequel.

$$\mathcal{L}_{ik} = \mathcal{L}_{i(k-1)} \oplus \{(w_{ik}, p_{ik})\}.$$

The states (w, p) in a list are sorted according to the decreasing value of p.

From the dynamic programming principle, dominated states, i.e. states (w, p) such that there exists a state (w', p') with $w' \leqslant w$ and $p' \geqslant p$, are removed from the list.

2.2. State elimination via upper bounds

In order to shrink lists $\mathcal{L}_{ik}, k \in \mathcal{N}_i$, an upper bound $\bar{z}(w,p)$, associated with state $(w,p) \in \mathcal{L}_{ik}$, is computed. For this purpose, we solve exactly the following linear continuous knapsack problem via the Martello & Toth's algorithm (see Martello & Toth, 1977):

$$(LP_i^{(w,p)}(C_i)) \begin{cases} \max p + \sum_{j=k+1}^{n_i} p_{ij}\cdot x_{ij}, \\ s.t. \sum_{j=k+1}^{n_i} w_{ij}\cdot x_{ij} \leqslant C_i - w, \\ x_{ij} \in [0,1], j \in \{k+1,\ldots,n_i\}. \end{cases} \tag{2.2}$$

Let $\bar{z}(w,p) = \lfloor z(LP_i^{(w,p)}(C_i)) \rfloor$.

If $\bar{z}(w,p) \leqslant \underline{z}(KSP)$, then the states (w, p) can be discarded.

We shall have:

$$\mathcal{L}_{ik} := \mathcal{L}_{i(k-1)} \oplus \{(w_{ik}, p_{ik})\} - \mathcal{D}_{ik} - \mathcal{B}_{ik},$$

where

- $\mathcal{D}_{ik}$ represents the list of dominated states in $\mathcal{L}_{i(k-1)} \oplus \{(w_{ik}, p_{ik})\}$,
- $\mathcal{B}_{ik}$ represents the list of states in $\mathcal{L}_{i(k-1)} \oplus \{(w_{ik}, p_{ik})\}$ to be eliminated via upper bounds.

In the sequel, this phase is the so-called *NextList* phase.

2.3. Fixing variables

The following two methods are used to fix variables of the problem (KP_i).

2.4. Variable reduction technique 1

Let $i \in \mathcal{M}, k \in N_i$ and $(w,p) \in \mathcal{L}_{ik}$.

If $p > \bar{z}(KSP)$, where $\bar{z}(KSP)$ is an upper bound of (KSP) obtained by the solution of the linear continuous relaxation of (KSP), then all free variables $x_{ij}, j \in \{k+1,\ldots,n_i\}$, can be fixed at 0 for the state (w,p).

Indeed, as $z(KSP) \leqslant \bar{z}(KSP)$, when $p > \bar{z}(KSP)$ we can stop the exploration of this state because it will not give a better optimal value for (KSP).

The second method to fix variables uses information provided by the solution of $(LP_i^{(w,p)}(C_i))$ associated to a state $(w,p) \in \mathcal{L}_{ik}$, $k \in N_i$. We use the following rule to fix the free variables of a state (w, p).

2.5. Variable reduction technique 2 (see Nemhauser & Wolsey, 1988)

Let $i \in \mathcal{M}, k \in N_i$ and $(w,p) \in \mathcal{L}_{ik}$.

Let d be the index of the critical variable of $(LP_i^{(w,p)}(C_i))$, i.e.:

$$\sum_{j=k+1}^{d-1} w_{ij} \leqslant C_i - w \text{ and } \sum_{j=k+1}^{d} w_{ij} > C_i - w.$$

If for $j \in \{k+1,\ldots d-1, d+1,\ldots,n_i\}$, we have

$$\left\lfloor z(LP_i^{(w,p)}(C_i)) - \left| p_{ij} - w_{ij}\cdot\frac{p_{id}}{w_{id}} \right| \right\rfloor \leqslant \underline{z}(KSP),$$

where $\underline{z}(KSP)$ is a lower bound of (KSP), then x_{ij} can be fixed to 1 if $j < d$ and to 0 otherwise. The computation of $\underline{z}(KSP)$ is detailed in the next section.

3. An algorithm for the KSP

In this section, we show how the dynamic programming method presented in the above section can be used to find an optimal solution of (KSP).

For simplicity of notation, $\underline{z}(KSP)$ is denoted by $\underline{z}$ in the sequel.

3.1. The main procedure DPKSP

The principle of our algorithm can be presented briefly as follows:

–A first lower bound of (KSP), $\underline{z}$, is computed with the greedy heuristic the so-called *GreedyKSP* (see Algorithm 1).

–At each step k of the dynamic programming method:

- the lists $(\mathcal{L}_{ik})_{i\in\mathcal{M},\ k\leqslant n_i}$ are computed via the procedure *NextList* presented in the above section,
- then, we try to reduce the free variables associated with each state in $(\mathcal{L}_{ik})_{i\in\mathcal{M},\ k\leqslant n_i}$,
- finally, the lower bound $\underline{z}$ and the capacities $(C_i)_{i\in\mathcal{M}}$, respectively, are updated via the procedures *UpdateZ* and *UpdateC*, respectively described below.

Please cite this article in press as: Boyer, V., et al. A dynamic programming method with lists for the knapsack sharing problem. *Computers & Industrial Engineering* (2010), doi:10.1016/j.cie.2010.10.015

– The dynamic programming phase is stopped when all the lists $(\mathcal{L}_{in_i})_{i\in\mathcal{M}}$ have been computed. These lists are used by the procedure *FindOptimalValue* to find an optimal solution for the (*KSP*).

Algorithm 1 (*GreedyKSP*).

For i from 1 to m do
  $p_i := 0$, $w_i := 0$ and $k_i := 1$
end do STOP:=0
while STOP=0 do
  $d := argmin\{p_1, p_2, \ldots, p_m\}$
  if $k_d \leqslant n_d$ then
    if $w_d + w_{dk_d} \leqslant C$, then
      x_{dk_d} is fixed to 1
      $p_d := p_d + p_{dk_d}$
      $w_d := w_d + w_{dk_d}$
    end if
      $k_d := k_d + 1$
    else
      STOP:=1
    end if
  end while
$\underline{z} = \min\{p_1, p_2, \ldots, p_m\}$.

The main procedure, *DPKSP*, to solve the KSP is given in Algorithm 2. The procedures *UpdateZ*, *UpdateC* and *FindOptimalValue* are detailed in the next sub-sections.

Algorithm 2 (*DPKSP*).

Initialisation:
$\underline{z} := \underline{z}(KSP) := GreedyKSP$
$\bar{z}(KSP) := \lfloor z(CKSP) \rfloor$
For i from 1 to m do
  $\mathcal{L}_{i0} := \{(0,0)\}$
end for
$k := 1$
$(C_i)_{i\in\mathcal{M}} := UpdateC(\underline{z}, (\mathcal{L}_{i0})_{i\in\mathcal{M}})$
Dynamic programming:
STOP := 0
while STOP = 0 do
  STOP := 1
  For i from 1 to m do
    If($k \leqslant n_i$)
      STOP := 0;
      $\mathcal{L}_{ik} := NextList(KP_i, \mathcal{L}_{i(k-1)})$
      For each state $(w,p) \in \mathcal{L}_{ik}$ do
        Try to fix the free variables
      end for
    end if
  end for
  $\underline{z} := UpdateZ(\underline{z}, (C_i)_{i\in\mathcal{M}}, (\mathcal{L}_{ik})_{i\in\mathcal{M}})$
  $(C_i)_{i\in\mathcal{M}} := UpdateC(\underline{z}, (\mathcal{L}_{ik})_{i\in\mathcal{M}})$
  $k := k + 1$
end while
Finding z^*:
$z(KSP) := FindOptimalValue(\underline{z}, (\mathcal{L}_{in_i})_{i\in\mathcal{M}})$

3.2. The procedure UpdateC

In this subsection, we present how the values of $(C_i)_{i\in\mathcal{M}}$ are updated. For this purpose, we compute the minimum values of the capacities $(C_i)_{i\in\mathcal{M}}$ resulting from an improvement of the current lower bound $\underline{z}$, of the KSP.

For $i \in \mathcal{M}$, and $k \in \mathcal{N}_i \cup \{0\}$, the following linear problem is associated with a state $(w,p) \in \mathcal{L}_{ik}$:

$$(\min W_i((w,p),\underline{z}))\begin{cases} \min w + \sum_{j=k+1}^{n_i} w_{ij}\cdot x_{ij}, \\ s.t.\ p + \sum_{j=k+1}^{n_i} p_{ij}\cdot x_{ij} \geqslant \underline{z}+1, \\ x_{ij} \in [0,1], j \in \{k+1,\ldots,n_i\}. \end{cases} \tag{3.1}$$

Let us define:

$$\min C_i(\mathcal{L}_{ik},\underline{z}) = \left\lfloor \min_{(w,p)\in\mathcal{L}_{ik}} \{z(\min W_i((w,p),\underline{z}))\} \right\rfloor.$$

If we want to improve the best lower bound, $\underline{z}$, then we must have, *for* $i \in \mathcal{M}$:

$$\sum_{j\in\mathcal{N}_i} w_{ij}\cdot x_{ij} \leqslant C - \sum_{i'\in\mathcal{M}-\{i\}} \min C_{i'}(\mathcal{L}_{i'k},\underline{z}),$$

with

$x_{ij} \in \{0,1\}$, for $j \in \mathcal{N}_i$.

For $i \in \mathcal{M}$, the initial value of C_i is given by

$$C_i = C - \sum_{i'\in\mathcal{M}-\{i\}} \min C_{i'}(\mathcal{L}_{i'0},\underline{z}).$$

At each step k of *DPKSP*, we try to improve the value of C_i with Algorithm 3.

Algorithm 3 (*UpdateC*).

For i from 1 to m do
  $C_i := C - \sum_{i'\in\mathcal{M}-\{i\}} \min C_{i'}(\mathcal{L}_{i'k},\underline{z})$
end for

3.3. The procedure UpdateZ

Rather than updating the lower bound $\underline{z}$, with the *GreedyKSP* heuristic, which is time consuming, we make use of all the lists $(\mathcal{L}_{ik})_{i\in\mathcal{M}}$ at step k in order to try to improve more efficiently this bound.

Indeed, for each state in the list, a local greedy heuristic can be used in order to select a particular state. The selected state of each list is then combined with other states so as to try to improve $\underline{z}$. The details of the heuristic are given in procedure *UpdateZ* (see Algorithm 4).

Algorithm 4 (*UpdateZ*).

For $i \in \mathcal{M}$ do $\mathcal{L}'_{ik} = \emptyset$
Greedy like step:
For i from 1 to m do
  For $(w,p) \in \mathcal{L}_{ik}$ do
    $W := w$ and $P := p$
    For j from $k+1$ to n_i do
      If $P \geqslant \underline{z}(KSP) + 1$ then
        exit the loop for
      end if
      If $W + w_{ij} \leqslant C_i$ then
        $W := W + w_{ij}$ and $P := P + p_{ij}$
      end if
  end for
  If $P \geqslant \underline{z}(KSP) + 1$ then
    $\mathcal{L}'_{ik} := \mathcal{L}'_{ik} \cup \{(W,P)\}$

(*continued on next page*)

Please cite this article in press as: Boyer, V., et al. A dynamic programming method with lists for the knapsack sharing problem. *Computers & Industrial Engineering* (2010), doi:10.1016/j.cie.2010.10.015

```
        end if
      end for
    end for
  Selected states:
  For i from 1 to m do
      Choose (W_i, P_i) ∈ L'_ik such that
      W_i := min_{(W,P)∈L'_ik} {W}
  end for
  Updating z(KSP):
  If Σ_{i∈M} W_i ⩽ C then
  z(KSP) := min_{i∈M} {P_i}
  end if
```

3.4. The procedure FindOptimalValue

In the end of the dynamic programming phase, all the lists $(\mathcal{L}_{in_i})_{i\in\mathcal{M}}$, are available. In this section, we show how these lists are combined in $\mathcal{O}(\sum_{i\in\mathcal{M}} C_i)$ in order to find the optimal value of (*KSP*).

The states (w, p) in a list are sorted according to the decreasing value of p. Due to the dominance principle, they are also sorted according to the decreasing value of w. Thus, if we want to check if a given bound $\bar{z} \geqslant \underline{z}$ is feasible, then we have to take in each list $\mathcal{L}_{in_i}, i \in \mathcal{M}$, the state (w^i, p^i) which satisfies:

$$w^i = \min\{w \mid p \geqslant \bar{z},\ (w,p) \in \mathcal{L}_{in_i}\}. \tag{3.2}$$

If $\sum_{i\in\mathcal{M}} w^i \leqslant C$, then we have found a better feasible bound for (*KSP*), i.e. $\bar{z}' = \min_{i\in\mathcal{M}}\{p^i\} \geqslant \bar{z} \geqslant \underline{z}$. Furthermore, all the states $(w,p) \in \mathcal{L}_{in_i}$ such that $p < p^i$ can be discarded. Otherwise, all the states $(w,p) \in \mathcal{L}_{in_i}$ such that $p > p^i$ (and $w > w^i$) can be removed as they will not provide a better solution. Indeed, in this case, we have the following inequalities:

$$z(KSP) < \bar{z} \leqslant p^i, \quad i \in \mathcal{M}.$$

Therefore, we have to decrease the value of the bound $\bar{z}$ and check if this new bound is feasible.

Algorithm 5 presents the procedure *FindOptimalValue*.

Algorithm 5 (*FindOptimalValue*).

```
  Initialization:
  For i from 1 to m do
      Let (w^i, p^i) be the first states in L_{in_i}
  end do
  z̄ := min_{i∈M} {p^i}
  z(KSP) := z(KSP) Checking feasibility:
  While z̄ > z(KSP) do
      For i from 1 to m do
          Find (w^i, p^i) ∈ L_{in_i} such that
          w^i := min{w | p ⩾ z̄, (w,p) ∈ L_{in_i}}
  end for
  If Σ_{i∈M} w^i ⩽ C then
          z(KSP) := z̄
          Exit the procedure
  Else
          For i from 1 to m do
              L_{in_i} := L_{in_i} − {(w,p) ∈ L_{in_i} | p > p^i}
          end for
          z̄ := min_{i∈M} max_{(w,p)∈L_{in_i}} {p | p < z̄}
      end if
  end while
```

We note that all the lists are considered only once in this procedure.

Table 1
Uncorrelated instances.

Group	n	m	Instances $1 \leqslant x \leqslant 4$
Am·*x*	1000	2–50	A02·*x*, A05·*x*, A10·*x*, A20·*x*, A30·*x*, A40·*x*, A50·*x*
Bm·*x*	2500	2–50	B02·*x*, B05·*x*, B10·*x*, B20·*x*, B30·*x*, B40·*x*, B50·*x*
Cm·*x*	5000	2–50	C02·*x*, C05·*x*, C10·*x*, C20·*x*, C30·*x*, C40·*x*, C50·*x*
Dm·*x*	7500	2–50	D02·*x*, D05·*x*, D10·*x*, D20·*x*, D30·*x*, D40·*x*, D50·*x*
Em·*x*	10,000	2–50	E02·*x*, E05·*x*, E10·*x*, E20·*x*, E30·*x*, E40·*x*, E50·*x*
Fm·*x*	20,000	2–50	F02·*x*, F05·*x*, F10·*x*, F20·*x*, F30·*x*, F40·*x*, F50·*x*

4. Computational experiments

The procedure *DPKSP* has been written in C and computational experiments have been carried out on an Intel Core 2 Duo T7500 (2.2 GHz).

We have used the set of problems of Hifi (see Hifi, 2005), see also (El Baz & Boyer, 2010) with 168 uncorrelated instances and 72 strongly correlated instances; each group of problems contains four instances. The problems are detailed in Tables 1 and 2. All the optimal values are known for these instances.

The average processing time for the four instances of each group of problems is given in Tables 3 and 4. The tables show that *DPKSP* is able to solve large instances (up to 20,000 variables and 50 classes) within reasonable computing time. In the correlated case, an optimal solution is obtained in less than 13 min. The processing time is less than 1.5 min in the uncorrelated case.

We note that our method is efficient when the number m of classes is relatively small (between 2 and 5 classes). This result can be explained by the fact that in this case we have to fill a limited number of knapsacks. The comparison with the results provided by Hifi and Sadfi (2002) and Hifi et al. (2005) shows the interest of our approach particularly in this context for both correlated and uncorrelated problems; indeed, the processing times of Hifi's algorithm tends to be important for these instances.

The memory occupancy is presented in Tables 5 and 6. Note that the vector x associated to each state is encoded as a bit string

Table 2
Correlated instances.

Group	n	m	Instances $1 \leqslant x \leqslant 4$
AmC·*x*	1000	2–10	A02C·*x*, A05C·*x*, A10C·*x*
BmC·*x*	2500	2–10	B02C·*x*, B05C·*x*, B10C·*x*
CmC·*x*	5000	2–10	C02C·*x*, C05C·*x*, C10C·*x*
DmC·*x*	7500	2–10	D02C·*x*, D05C·*x*, D10C·*x*
EmC·*x*	10,000	2–10	E02C·*x*, E05C·*x*, E10C·*x*
FmC·*x*	20,000	2–10	F02C·*x*, F05C·*x*, F10C·*x*

Table 3
Uncorrelated instances: time processing (s).

Inst.	*t.*	Inst.	*t.*	Inst.	*t.*
A02·*x*	0.02	C02·*x*	0.21	E02·*x*	3.02
A05·*x*	0.15	C05·*x*	1.96	E05·*x*	5.55
A10·*x*	0.14	C10·*x*	4.02	E10·*x*	15.24
A20·*x*	0.06	C20·*x*	4.55	E20·*x*	18.38
A30·*x*	0.01	C30·*x*	2.53	E30·*x*	13.27
A40·*x*	0.00	C40·*x*	0.85	E40·*x*	13.24
A50·*x*	0.01	C50·*x*	0.66	E50·*x*	15.26
B02·*x*	0.16	D02·*x*	0.66	F02·*x*	3.00
B05·*x*	0.69	D05·*x*	4.39	F05·*x*	35.16
B10·*x*	1.20	D10·*x*	6.94	F10·*x*	46.31
B20·*x*	0.69	D20·*x*	10.83	F20·*x*	68.89
B30·*x*	0.01	D30·*x*	8.22	F30·*x*	67.21
B40·*x*	0.01	D40·*x*	7.02	F40·*x*	78.46
B50·*x*	0.01	D50·*x*	4.02	F50·*x*	78.86

Please cite this article in press as: Boyer, V., et al. A dynamic programming method with lists for the knapsack sharing problem. *Computers & Industrial Engineering* (2010), doi:10.1016/j.cie.2010.10.015

Table 4
Correlated instances: time processing (s).

Inst.	*t.*	Inst.	*t.*	Inst.	*t.*
A02C·*x*	1.18	C02C·*x*	18.48	E02C·*x*	59.44
A05C·*x*	1.32	C05C·*x*	26.25	E05C·*x*	149.29
A10C·*x*	1.72	C10C·*x*	33.28	E10C·*x*	164.33
B02C·*x*	9.61	D02C·*x*	53.40	F02C·*x*	642.84
B05C·*x*	10.08	D05C·*x*	53.61	F05C·*x*	724.81
B10C·*x*	9.63	D10C·*x*	98.14	F10C·*x*	598.05

Table 5
Uncorrelated instances: memory occupancy (Mo).

Instance	DPKSP	ALGO (HIFI)
Am·*x*	<0.01	1.59
Bm·*x*	0.05	9.95
Cm·*x*	0.11	38.15
Dm·*x*	0.33	89.4
Em·*x*	0.41	158.79
Fm·*x*	1.01	636.95

Table 6
Correlated instances: memory occupancy (Mo).

Instance	DPKSP	ALGO (Hifi)
AmC·*x*	0.10	1.60
BmC·*x*	0.30	9.97
CmC·*x*	0.68	39.76
DmC·*x*	1.26	89.60
EmC·*x*	1.96	159.23
FmC·*x*	6.21	638.42

of 32 bits. These tables compare DPKSP with the algorithm of Hifi whose space complexity is $\mathcal{O}(n \cdot C)$. We see that the approach of Hifi needs for some instances a large amount of memory (up to 638.42 Mo) contrarily to our method (<7 Mo). Indeed, the space complexity of the dynamic programming algorithm with lists for a knapsack problem of n variables with the capacity C is $\mathcal{O}(min\{2^{n+1}, nC\})$ (see Martello & Toth, 1990). The space complexity of DPKSP is $\mathcal{O}\left(\sum_{i=1}^{m} min\{2^{n_i+1},\ n_i \cdot C_i\}\right)$. Thus it is bounded by $\mathcal{O}(n \cdot C)$. DPKSP is able to solve large problems with limited memory occupancy.

5. Conclusions and perspectives

In this paper, we have proposed a method to solve the KSP with a dynamic programming list algorithm.

The original problem (*KSP*) is decomposed into a set of knapsack problems. The initial value of the knapsack capacity is obtained via an overestimation. This value is updated and decreased throughout the solution process.

Computational results show that best results were obtained when the number of classes is relatively small. They show also that *DPKSP* is able to solve large instances within reasonable computing time and small memory occupancy. Moreover, our method use less memory than previous methods in the literature.

We think that very good results can be obtained with our method for instances with bigger capacities and a great number of classes (in the problems considered, capacities are generated so that they are equal to the half sum of all item weights, as a consequence, the bigger the weights, the bigger the capacity is).

In future work, it would also be interesting to consider a multi method that combines the advantages of our approach with the one of Hifi et al.

In order to decrease the processing time, parallel implementation of *DPKSP* could be considered. Indeed, the computations on knapsack sub-problems are independent and could be done in parallel.

Acknowledgements

The authors would like to thank the reviewers for their helpful comments contributing to improve the presentation of the paper.

References

Bellman, R. (1957). *Dynamic programming*. Princeton, NJ: Princeton University Press.
Boyer, V., El Baz, D., & Elkihel, M. (2009). Heuristics for the 0–1 multidimensional knapsack problem. *European Journal of Operational Research, 199*(3), 658–664.
Boyer, V., El Baz, D., & Elkihel, M. (2010). Solution of multidimensional knapsack problems via cooperation of dynamic programming and branch and bound. *European Journal of Industrial Engineering, 4*(4), 434–449.
Brown, J. R. (1979). The knapsack sharing problem. *Operations Research, 27*, 341–355.
Brown, J. R. (1991). Solving knapsack sharing with general tradeoff functions. *Mathematical Programming, 51*, 55–73.
Brown, J. R. (1994). Bounded knapsack sharing. *Mathematical Programming, 67*, 343–382.
El Baz, D., Boyer, V. (2010). Library of instances for knapsack sharing problems. <http://www.laas.fr/CDA/23-31298-Knapsack-sharing-problems.php>.
El Baz, D., & Elkihel, M. (2005). Load balancing methods and parallel dynamic programming algorithm using dominance technique applied to the 0–1 knapsack problem. *Journal of Parallel and Distributed Computing, 65*, 74–84.
Hifi, M., M'Halla, H., & Sadfi, S. (2005). An exact algorithm for the knapsack sharing problem. *Computers and Operations Research, 32*, 1311–1324.
Hifi, M., & Sadfi, S. (2002). The knapsack sharing problem: An exact algorithm. *Journal of Combinatorial Optimization, 6*, 35–54.
Hifi, M. Library of instances. (2005) <ftp://cermsem.univ-paris1.fr/pub/CERMSEM/hifi/KSP>.
Martello, S., & Toth, P. (1977). An upper bound for the zero–one knapsack problem and a branch and bound algorithm. *European Journal of Operations Research, 1*, 69–175.
Martello, S., & Toth, P. (1990). *Knapsack problems: Algorithms and computer implementation*. New York: John Wiley & Sons.
Nemhauser, G. L., & Wolsey, L. A. (1988). *Integer and combinatorial optimization*. New York: Wiley.
Yamada, T., Futakawa, M., & Kataoka, S. (1997). Heuristic and reduction algorithms for the knapsack sharing problem. *Computers and Operations Research, 24*, 961–967.
Yamada, T., Futakawa, M., & Kataoka, S. (1998). Some exact algorithms for the knapsack sharing problem. *European Journal of Operational Research, 106*, 177–183.

Please cite this article in press as: Boyer, V., et al. A dynamic programming method with lists for the knapsack sharing problem. *Computers & Industrial Engineering* (2010), doi:10.1016/j.cie.2010.10.015