www.elsevier.com/locate/caor

A dynamic programming algorithm for the Knapsack Problem with Setup

Khalil Chebil, Mahdi Khemakhem

PII: S0305-0548(15)00121-5
DOI: http://dx.doi.org/10.1016/j.cor.2015.05.005
Reference: CAOR3786

To appear in: *Computers & Operations Research*

Received date: 27 May 2014
Revised date: 10 May 2015
Accepted date: 11 May 2015

Cite this article as: Khalil Chebil, Mahdi Khemakhem, A dynamic programming algorithm for the Knapsack Problem with Setup, *Computers & Operations Research,* http://dx.doi.org/10.1016/j.cor.2015.05.005

A dynamic programming algorithm for the Knapsack Problem with Setup

Khalil Chebil

khalil.chebil@issatm.rnu.tn
LOGIQ, University of Sfax - Tunisia

Mahdi Khemakhem

mahdi.khemakhem@enetcom.rnu.tn
LOGIQ, University of Sfax - Tunisia

Abstract

The Knapsack Problem with Setup (KPS) is a generalization of the classical Knapsack problem (KP), where items are divided into families. An individual item can be selected only if a setup is incurred for the family to which it belongs. This paper provides a dynamic programming (DP) algorithm for the KPS that produces optimal solutions in pseudo-polynomial time. In order to reduce the storage requirements of the algorithm, we adopt a new technique that consists in converting a KPS solution to an integer index. Computational experiments on randomly generated test problems show the efficiency of the DP algorithm compared to the ILOG's commercial product CPLEX 12.5.

Keywords: Knapsack problems, setup, dynamic programming, combination, production planning

1. Introduction

We will refer to the Knapsack Problem with Setup as KPS. It is described as a knapsack problem with additional fixed setup costs discounted both in the objective function and in the constraints. This problem is particularly prevalent in production planning applications where resources need to be set up before a production run.

Our interest in this model was originally motivated by practical problems at a production project with SOTUVER, a leading manufacturer and supplier of hollow glass in the Tunisian agro-alimentary glass packing industry. This company produces several types of products, including bottles, flacons, and pots. The most important phase in the manufacturing process, is the phase of shaping. In fact, to change the production from one product family to an another, the production machinery must be set up and moulds must be changed in the moulding machine. There is no setup between products in the same family. These changes in the manufacturing process require significant setup time and costs. Accordingly, the company needs to decide on how to choose orders so as to maximize the total profit. This represents a typical case involving a knapsack problem with setup model that can be used to solve this problem.

The Knapsack Problem with Setup is defined by a knapsack capacity $b \in \mathbb{N}$ and a set of N classes of items. Each class $i \in \{1, \ldots, N\}$ is composed of n_i items and characterized by a negative integer f_i and a non-negative integer d_i representing its setup cost and setup capacity consumption, respectively. Each item $j \in \{1, \ldots, n_i\}$ of a class i is labeled by a profit $c_{ij} \in \mathbb{N}^{N \times n_i}$ and a capacity consumption $a_{ij} \in \mathbb{N}^{N \times n_i}$. The aim is to maximize the total profit of the selected items minus the fixed costs incurred for setting-up the selected classes.

The KPS can be formulated by a 0-1 linear program as follows:

$$\text{Max } z = \sum_{i=1}^{N} \sum_{j=1}^{n_i} c_{ij} x_{ij} + \sum_{i=1}^{N} f_i y_i \tag{1}$$

s.t.

$$\sum_{i=1}^{N} \sum_{j=1}^{n_i} a_{ij} x_{ij} + \sum_{i=1}^{N} d_i y_i \leq b \tag{2}$$

$$x_{ij} \leq y_i \qquad \forall i \in \{1, \ldots, N\}, \forall j \in \{1, \ldots, n_i\} \tag{3}$$

$$x_{ij}, y_i \in \{0, 1\} \qquad \forall i \in \{1, \ldots, N\}, \forall j \in \{1, \ldots, n_i\} \tag{4}$$

Equation (1) represents the objective function for KPS. Constraint (2) ensures that the weight of selected items in the knapsack, including their setup capacity consumption, does not exceed knapsack capacity b. Constraints (3) guarantee

that each item is selected only if it belongs to a class that has been setup. Constraints (4) require the decision variables to be binary where x_{ij} represents the item variables and y_i the class setup variables. In fact, y_i is equal to 1 if the knapsack is set up to accept items belonging to class i; otherwise it is equal to 0. x_{ij} is equal to 1 if the item j of the class i is placed in the knapsack and is equal to 0 otherwise.

The standard 0-1 Knapsack Problem (KP), which is known to be an NP-hard problem [1], is a variant of KPS wherein the number of item classes is equal to one. The KP has been the subject of extensive research, and a detailed summary of the major works on the issue is available in [1] and [2].

KPS has similarities to several other knapsack variants in addition to KP. As presented by Chajakis and Guignard [3], the Set-up Knapsack Problem (SKP) occurs as a subproblem of a parallel machine scheduling problem with setups. This problem is similar to KPS, except for the fact that the setup cost of each class and the profit associated to each item can take on real numbers that may be negative or non-negative. A further constraint is added to make sure that if the knapsack is setup for a class, at least one item of this class is selected.

A thorough survey of the literature on two variants of knapsack problems with setups has recently been presented in the work of Michel et al. [4] namely the multiple-class binary knapsack problem with setups (MBKPS) where item weights are assumed to be a multiple of there class weight and the continuous knapsack problems with setups (CKS) where each class holds a single item and a fraction of an item can be selected while incurring a full setup. Michel et al. provided an extension of the branch-and-bound algorithm proposed by Horowitz and Sahni [5] for problems with positive setup costs.

The Fixed Charge Knapsack Problem (FCKP) is a particular case of KPS that includes the setup capacity consumption but not the setup cost, see [6]. The bounded Knapsack Problem with Setups (BKPS), which is a generalization of FCKP wherein a limited copy of each item may be added to the knapsack, was presented in [7] and [8]. The Integer Knapsack Problem with Set-up Weights (IKPSW), see [9], is a generalization of BKPS in which an unlimited copy of

each item may be added to the knapsack. This model has found application in the area of aviation security, see [10] and [11].

The remainder of this paper is organized as follows: In section 2, we introduce a basic dynamic programming (DPB) algorithm to optimally solve the KPS. We then present an improved version of the DPB algorithm (DPI) that has the ability to solve large problem instances (up to 10000 items) in a pseudo-polynomial time. In section 3, we analyze the computational behavior of these algorithms on a set of randomly generated test problems. Section 4 concludes by a brief summary and directions for future research.

2. A dynamic programming algorithm for the KPS

Dynamic Programming (DP) was introduced by Bellman [12]. The basic idea of DP is to solve a problem by a divide-and-conquer approach wherein the solutions of overlapping subproblems are reused to avoid the recalculation of solutions. Toth [13] and Horowitz and Sahni [5] presented an improved dynamic programming algorithm for solving the knapsack problem. More recently, Pisinger [14] proposed a minimal algorithm for the 0-1 knapsack problem based on dynamic programming. A first hybrid approach combining DP with an enumerative scheme was proposed by Plateau and Elkihel [15]. Generally, dynamic programming-based algorithms are efficient and easy to implement particularly for small and medium-sized instances. Chajakis and Guignard [3] suggested a dynamic programming algorithm and two versions of a two-phase enumerative scheme for the SKP. Experiments were performed only on uncorrelated instances with low range coefficients (i.e. a_{ij} from [1,10]). Since the DP algorithm presented in their paper has a pseudo-polynomial worst case complexity, the large number of coefficients would increase the difficulty of instances and need more storage. In their model, the setup cost of each class and the profit associated with each item can take on real numbers that may be negative or non-negative. The fact that parts of the variables are fixed to 0 by a preprocessing procedure remarkably reduces the size of the problem, thus making instances easier.

McLay and Jacobson [9] have also provided a dynamic programming algorithm for IKPSW that produces an optimal solution in a pseudo-polynomial time but is not practical for solving large problem instances.

In this section, we start by providing a brief outline of a DP algorithm for the 0-1 knapsack problem. We then present a number of preliminary considerations in the knapsack problem with setup that are relevant to the design of the proposed algorithm. We also describe the general DP procedure.

2.1. A dynamic programming algorithm for the 0-1 KP

The classical knapsack problem is defined as follows: we consider a set of n items, with each item j having an integer profit c_j and an integer weight a_j. The problem is to choose a subset of the items such that their overall profit is maximized while the overall weight does not exceed a given capacity b. We can formulate the model as the following binary programming model:

$[KP]$:

$$\text{Max } z = \sum_{j=1}^{n} c_j x_j \tag{5}$$

s.t.

$$\sum_{j=1}^{n} a_j x_j \leq b \tag{6}$$

$$x_j \in \{0,1\} \qquad \forall j \in \{1,\ldots,n\} \tag{7}$$

where the binary decision variables x_j are used to indicate whether item j is included in the knapsack or not.

Given a pair of integers k $(1 \leq k \leq n)$ and β $(1 \leq \beta \leq b)$, we consider the sub-instance of KP which consist of the first k items and a capacity β. Let V_β^k denote its optimal solution value:

$$V_\beta^k = \{\sum_{j=1}^{k} c_j x_j : \sum_{j=1}^{k} a_j x_j \leq \beta, \quad x_j \in \{0,1\} \quad \forall j \in \{1,\ldots,n\}\} \tag{8}$$

We initially have:

$$V_\beta^1 = \begin{cases} 0 & \text{for} \quad 0 \leq \beta < a_1 \\ c_1 & \text{for} \quad a_1 \leq \beta < b \end{cases}$$

Dynamic programming consists in considering n stages (for k increasing from 1 to n) and computing at each stage $k > 1$ the values V_β^k (for β increasing from 0 to b) using the classical Bellman recursion [12]:

$$V_\beta^k = \begin{cases} V_\beta^{k-1} & \text{for} \quad 0 \leq \beta < a_k \\ \max\left(V_\beta^{k-1}, V_{\beta-a_k}^{k-1} + c_k\right) & \text{for} \quad a_k \leq \beta < b \end{cases}$$

We call states S_β^k the feasible solutions corresponding to the V_β^k values. The optimal solution of the problem is the state S_b^n corresponding to V_b^n. Similarly, the KPS can be solved using the same dynamic programming algorithm.

2.2. Preliminary considerations

Unlike the KP, items are disjoint in different classes in the KPS. So, we consider $n^* = \sum_{i=1}^{n} n_i$ the total number of items and, throughout this section, we assume that the n^* items are arranged in contiguous blocks.

Let's, for example, consider $n^* = 9$ items as divided into 3 families, $N_1 = \{1, 2, 3\}$, $N_2 = \{4, 5, 6, 7\}$ and $N_3 = \{8, 9\}$, with $n_1 = 3$, $n_2 = 4$ and $n_3 = 2$, respectively. We consider $t(k)$ the family index of item k, such that $t(1) = t(2) = t(3) = 1$, $t(4) = t(5) = t(6) = t(7) = 2$, and $t(8) = t(9) = 3$. Let's also consider $n(k) = k - \sum_{i=1}^{t(k)-1} n_i$ as the index of item k in its family $t(k)$. Therefore, according to the previous example, we have $n(5) = 5 - \sum_{i=1}^{2-1} n_i = 5 - n_1 = 5 - 3 = 2$. In fact, the item with index 5 corresponds to the second item in the second class.

We define the problem $KPS(k, \beta)$ as a sub-instance of KPS. $KPS(k, \beta)$ is a Setup Knapsack with capacity $\beta \in \{0, \dots b\}$ wherein only the first $k \in K = \{1, \dots, n^*\}$ items can be selected to be included in the knapsack.

$[KPS(k,\beta)]$:

$$\text{Max} \sum_{i=1}^{t(k)-1}\sum_{j=1}^{n_i} c_{ij}x_{ij} + \sum_{j=1}^{n(k)} c_{t(k)j}x_{t(k)j} + \sum_{i=1}^{t(k)} f_i y_i \tag{9}$$

$$s.t.$$

$$\sum_{i=1}^{t(k)-1}\sum_{j=1}^{n_i} a_{ij}x_{ij} + \sum_{j=1}^{n(k)} a_{t(k)j}x_{t(k)j} + \sum_{i=1}^{t(k)} d_i y_i \leq \beta \tag{10}$$

$$x_{ij} \leq y_i \qquad \forall i \in \{1,\ldots,t(k)\} \tag{11}$$
$$\forall j \in \{1,\ldots,n_i\}$$

$$x_{ij}, y_i \in \{0,1\} \qquad \forall i \in \{1,\ldots,t(k)\} \tag{12}$$
$$\forall j \in \{1,\ldots,n_i\}$$

Consider two auxiliary problems, namely $KPS(k,\beta)^0$ and $KPS(k,\beta)^1$, which are derived from $KPS(k,\beta)$ as follows:

$$KPS(k,\beta)^0 = (KPS(k,\beta)|y_{t(k)} = 0),$$

$$KPS(k,\beta)^1 = (KPS(k,\beta)|y_{t(k)} = 1).$$

Let $(V_\beta^k)^0$ and $(V_\beta^k)^1$ denote the optimal values of $KPS(k,\beta)^0$ and $KPS(k,\beta)^1$, respectively.

2.3. The basic dynamic programming algorithm

The dynamic programming algorithm labeled DPB (see Appendix A algorithm 1) consists of n^* stages. At each stage $k \in K$, we compute $(V_\beta^k)^0$ and $(V_\beta^k)^1$ for each value of $\beta \in \{0,\ldots,b\}$. Therefore,there are $b+1$ discrete states at each stage k. We call states $(S_\beta^k)^0$ and $(S_\beta^k)^1$ the feasible solutions corresponding to the $(V_\beta^k)^0$ and $(V_\beta^k)^1$ values, respectively. An optimal solution S_{opt} of KPS corresponds to the maximum of $(V_b^{n^*})^0$ and $(V_b^{n^*})^1$.

At the initial stage, $k = 1$ we set:

$$(V_\beta^1)^1 = \begin{cases} -\infty & \text{for} \quad 0 \leq \beta < a_{t(1),1} + d_{t(1)} \\ c_{t(1),1} + f_1 & \text{for} \quad a_{t(1),1} + d_{t(1)} \leq \beta < b \end{cases}$$

$$(V_\beta^1)^0 = 0 \quad \text{for} \quad 0 \leq \beta \leq b$$

At stage $k > 1$, if $t(k) = t(k-1)$, the recursive equations are:

$$(V_\beta^k)^1 = \begin{cases} (V_\beta^{k-1})^1 & \text{for} \quad 0 \le \beta < a_{t(k),n(k)} + d_{t(k)} \\ \max \begin{pmatrix} (V_\beta^{k-1})^1, \\ (V_{\beta - a_{t(k),n(k)}}^{k-1})^1 + c_{t(k),n(k)}, \\ (V_{\beta - a_{t(k),n(k)} - d_{t(k)}}^{k-1})^0 + c_{t(k),n(k)} + f_{t(k)} \end{pmatrix}, & \text{for} \quad a_{t(k),n(k)} + d_{t(k)} \le \beta < b \end{cases}$$

$$(V_\beta^k)^0 = (V_\beta^{k-1})^0 \quad \text{for} \quad 0 \le \beta \le b$$

At stage $k > 1$, if $t(k) \neq t(k-1)$, the recursive equations are:

$$(V_\beta^k)^1 = \begin{cases} -\infty & \text{for} \quad 0 \le \beta < a_{t(k),n(k)} + d_{t(k)} \\ \max \begin{pmatrix} (V_{\beta - a_{t(k),n(k)} - d_{t(k)}}^{k-1})^1 + c_{t(k),n(k)} + f_{t(k)}, \\ (V_{\beta - a_{t(k),n(k)} - d_{t(k)}}^{k-1})^0 + c_{t(k),n(k)} + f_{t(k)} \end{pmatrix}, & \text{for} \quad a_{t(k),n(k)} + d_{t(k)} \le \beta < b \end{cases}$$

$$(V_\beta^k)^0 = \max\left((V_\beta^{k-1})^0, (V_\beta^{k-1})^1\right) \text{ for } 0 \le \beta \le b$$

The DPB algorithm finds the optimal value of KPS in a pseudo-polynomial $O(b \times n^*)$ time. Its storage requirements are also $O(b \times n^*)$. However, due to large storage requirements, the DPB algorithm could not solve large instances with high knapsack capacity. This limitation calls for the design of a new space reduction technique.

2.4. An improved dynamic programming algorithm

The basic idea of the space reduction technique is to represent a solution of the KPS in a different way. In this section, we make use of some KPS specifications to reduce the number of variables in the representation of a KPS solution, and then present a new technique that consists in converting the reduced KPS solution to an integer index.

2.4.1. Some KPS specifications

As mentioned earlier, the Knapsack Problem is a special case of the KPS (where $n = 1$). The special structure of KPS allows us to fix the setup variables (y_i=0 or y_i=1) to transform the KPS into a KP. Let's consider a set of item

classes $Y^1 = \{i \in \{1, \ldots, N\}/y_i = 1\}$ where the knapsack is set up to accept items belonging in each class in Y^1 and consider the complementary set of item classes $Y^0 = \{i \in \{1, \ldots, N\}/y_i = 0\}$. We define the problem $KPS[Y]$ ($Y = Y^0 \cup Y^1$) to be a sub-problem of KPS. $KPS[Y]$ is a knapsack problem with capacity $b - \sigma$ and an objective function minimized by a negative integer setup cost δ. Then, the $KPS[Y]$ can be formulated by a KP 0-1 linear program as follows:

$$\text{Max } z = \sum_{i \in Y^1} \sum_{j=1}^{n_i} c_{ij} x_{ij} + \delta \tag{13}$$

s.t.

$$\sum_{i \in Y^1} \sum_{j=1}^{n_i} a_{ij} x_{ij} \leq b - \sigma \tag{14}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i \in Y^1, \forall j \in \{1, \ldots, n_i\} \tag{15}$$

where

$$\delta = \sum_{i \in Y^1} f_i \text{ and } \sigma = \sum_{i \in Y^1} d_i$$

With these considerations, we only fix the setup variables y_i to 1 or 0, and we use an exact method to solve a knapsack problem with the remaining ones, including the item variables x_{ij} leading to the optimal solution. In this way, if we find the optimal setup variable combination Y^*, we can easily find the optimal solution of KPS by optimally solving the knapsack problem $KPS[Y^*]$ with CPLEX.

A KPS solution can be represented by two sets: a set of item variables $X = \{x_{ij}, i = 1, \ldots, N; j = 1, \ldots, n_i\}$ and a set of setup variables $Y = \{y_i, i = 1, \ldots, N\}$. Consider an example of a KPS instance defined by:

$$N = 3, b = 90, [n_i, i = 1, \ldots, 3] = [4, 3, 3],$$

$$[f_i, i = 1, \ldots, 3] = [-10, -13, -8], [d_i, i = 1, \ldots, 3] = [6, 5, 7],$$

$$[c_{ij}, i = 1, \ldots, 3; j = 1, \ldots, n_i] = \begin{bmatrix} 20 & 24 & 19 & 23 \\ 26 & 22 & 26 & \\ 25 & 24 & 29 & \end{bmatrix},$$

$$[a_{ij}, i = 1, \ldots, 3; j = 1, \ldots, n_i] = \begin{bmatrix} 15 & 19 & 14 & 18 \\ 17 & 17 & 21 & \\ 20 & 19 & 24 & \end{bmatrix}.$$

The optimal solution to this example is $X^* = \{\overbrace{0,0,0,0}^{class1}, \overbrace{1,1,0}^{class2}, \overbrace{1,0,1}^{class3}\}$, $Y^* = \{0,1,1\}$, with an objective value $z = 81$. We can see that the knapsack is set up to accept only items from classes 2 and 3. Thus, all item variables belonging to class 1 are equal to 0. To obtain the set X^*, we just use CPLEX to optimally solve KPS$[Y^*]$ which, in this example, is a knapsack problem of 6 items (items belonging in class 2 and 3), with a capacity $b' = b - \sigma = 90 - 5 - 7 = 78$ and an initial value $z = \delta = -13 - 8 = -21$. In the following section, we consider only the set Y to represent a KPS solution.

2.4.2. Solution representation

In order to reduce the storage requirement, we adopt a new space reduction technique wherein a solution is converted to an integer index. In fact, the solution $S = (0,1,1,0,1) = \{2,3,5\}$ represents a solution with $N = 5$ classes (where classes are indexed from 1 to 5). We can consider S as a $< 3\text{-}combinations >$ of Y. Combinations can refer to the combination of N things k taken at a time with or without repetitions. In our case, repetitions were not allowed. The number of $< k\text{-}combinations >$ is equal to the binomial coefficient $\binom{N}{k} = \frac{N!}{k!(N-k)!}$ (also denoted by C_k^N). Figure 1 presents the list of all $< 3\text{-}combinations >$ of 5 classes. It can be noted that S represents a solution with index 3. Thus, all $< k\text{-}combinations >$ of Y can be put in bijection with the natural numbers from 0 to $Binomial(N,k) - 1$. Algorithm 2 (see Appendix A) is used to compute $\binom{N}{k}$, algorithm 3 (see Appendix A) provides the index of a given combination S and algorithm 4 (see Appendix A) provides the combination that corresponds to an index for a given N and k.

Figure 2 represent a sketch of the improved dynamic programming (DPI) algorithm. It consists of n^* stages. In every stage k of the DPB algorithm, we have to store a set of b solutions (S_β^k) which takes $O(b \times n^*)$ space. How-

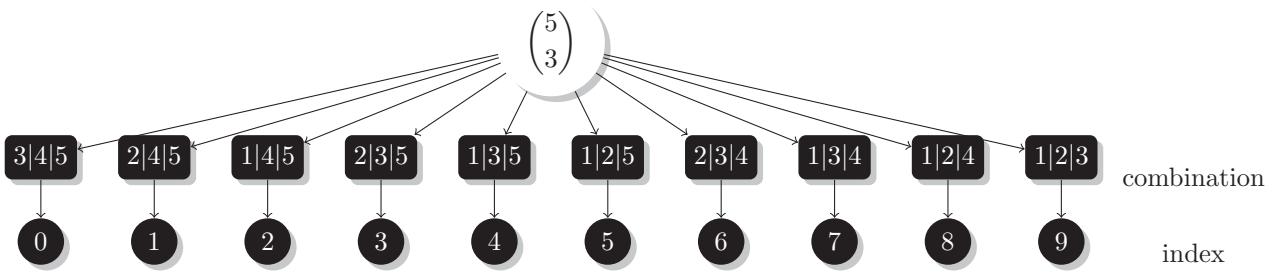

Figure 1: $<$ 3-*combinations* $>$ list of 5

ever, using the new space reduction technique that converts a KPS solution to an integer index allow us to store only the index associated to each solution which reduce considerably the storage requirement for each stage to $O(b)$. This algorithm provides a KPS solution Y^* with only setup variables y_i. To obtain the complete optimal KPS solution, we fix setup variables according to Y^* and then optimally solve the knapsack problem $KPS[Y^*]$ using CPLEX. The detailed algorithm is presented in Algorithm 5 (see Appendix A).

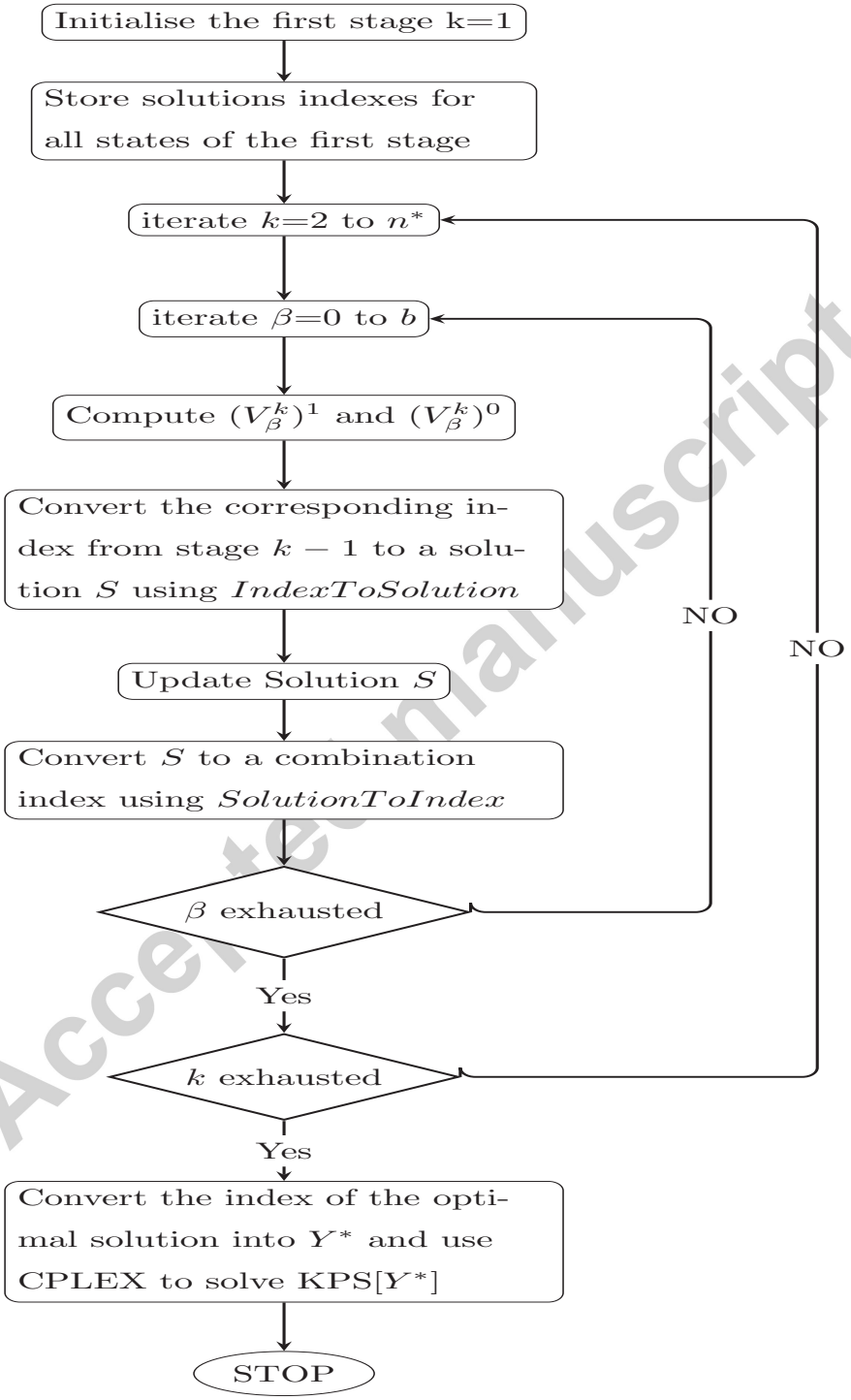

Figure 2: A sketch of the DPI algorithm

3. Experimental Results

In this section, we present the results of the DPI algorithm in relation to some instances of the knapsack problem with setup. The algorithm has been implemented in *c language*, and run on a 2.1 GHZ *intel core*$^{TM}i3$ with 2 GB of memory.

Chajakis and Guignard [3] presented a dynamic programming algorithm and two versions of a two-phase enumerative scheme consisting in converting the problem into a multiple choice knapsack problem (MCKP). Branch-and-Bound (MCBB) and dynamic programming (MCDP) are both used to solve the MCKP. Computational results demonstrates that MCDP and MCBB could not solve problems with more than 1000 and 2000 variables respectively, while DP was able to solve problems with 4000 variables. Also, the DP algorithm outperforms both versions of the enumerative scheme in term of storage, robustness and speed. for these reasons, we compared our improved DP to a basic dynamic programming algorithm and CPLEX 12.5.

Due to the unavailability of benchmark instances in the literature, we tested the algorithm on a set of randomly generated instances with a total number of items $n^* \in \{500, 1000, 2500, 5000, 10000\}$, $N \in \{5, 10, 20, 30\}$, and a knapsack capacity b up to 250 000 (available at https://sites.google.com/site/chebilkh/knapsack-problem-with-setup). This set is a collection of 200 strongly correlated instances (10 instances are generated for each n^*-N). We designed a random generation scheme, as presented in [3] and [16], which makes use of the following rules:

- Setup cost and capacity consumption is given by

$$f_i = -e_1(\sum_{j=1}^{n_i} c_{ij}) \tag{16}$$

$$d_i = -e_1(\sum_{j=1}^{n_i} a_{ij}) \tag{17}$$

  where e_1 is uniform from [0.15, 0.25].

- a_{ij} randomly generated in [10, 100].

- $c_{ij} = a_{ij} + 10$.
- $b = 0.5 * \sum_{i=1}^{N} \sum_{j=1}^{n_i} a_{ij}$.
- The cardinality of each class n_i, for $i = 1, \ldots, N$, is in $[k - \frac{k}{10}, k + \frac{k}{10}]$ with $k = \frac{n^*}{N}$.

It is worth noting that CPLEX 12.5 finds an optimal solution for 79 out of 200 problems with a limit of 1 hour of CPU Time. For the remaining problems, CPLEX terminates with error (exceeds the capacity of RAM memory or exceeds the limit of CPU time). The results for those 200 randomly generated instances are presented in detail in table B.3 (see Appendix B).

Table 1 summarizes the results obtained by the DPI algorithm as compared to those generated by CPLEX 12.5 and DPB when solving KPS. Each row summarizes 10 instances. The first two columns present the total number of items n^* and the number of classes N. The next five columns show the corresponding sum of values provided by the DPI algorithm (DPI_{obj}), the sum of values provided by CPLEX ($CPLEX_{obj}$), the total running time of the DPI algorithm (DPI_{cpu}), the total running time of the DPB algorithm (DPB_{cpu}), and the total running time of CPLEX ($CPLEX_{cpu}$). Processing times (CPU's) are reported in seconds, the last column represents the average of the gaps that are calculated as follows:

$$GAP(\%) = \frac{100 \times (CPLEX_{obj} - DPI_{obj})}{CPLEX_{obj}}$$

The results illustrated in table 1 show the limited capacity of the DPB algorithm with regard to solving large instances (up to 1000 items). In fact, the use of large coefficients a_{ij} increases the difficulty of instances and storage requirement. It can also be noted that the new space reduction technique allowed the DPI algorithm to solve large instances (up to 10 000 items) in a very short computation time compared to CPLEX. Furthermore, we note that the DPI algorithm finds an optimal solution for all 200 problems (negative value of the gap indicates an improvement of the best solution provided by CPLEX).

The results graphically displayed in figure 3 illustrate a comparison in terms of CPU time between the DPI algorithm, DPB algorithm, and CPLEX, where the x axis represents the set of 200 problems while the y axis represents the CPU time needed to solve each problem. It can be clearly observed that the DPI algorithm was much more faster than CPLEX in most of instances.

In order to properly evaluate the performance of the DPI algorithm, we report in table 2 the results obtained by the DPI algorithm compared to CPLEX 12.5. Results are means over a set of randomly generated test problems with a total number of items $n^* = 10000$. This set is a collection of 40 weekly correlated instances (10 instances are generated for each $N \in \{5, 10, 20, 30\}$), where a_{ij} is randomly generated in [0,100] and $c_{ij} = a_{ij}+r$ (r randomly generated in [10,50]). The first column presents the number of classes N. The next two columns show the corresponding total running time of the DPI algorithm and CPLEX. The last column represent the number of instances where CPLEX finds an optimal solution with a CPU time limit of 3600 seconds for each problem. Experiments have shown that DPI dominates CPLEX in term of robustness and speed for strongly and weekly correlated instances.

Table 1: Performance of DPI algorithm compared to DPB and CPLEX 12.5

n^*	N	DPI_{obj}	$CPLEX_{obj}$	DPI_{cpu} (s)	DPB_{cpu} (s)	$CPLEX_{cpu}$ (s)	GAP (%)
500	5	110738	110738	3,073	347,711	2225,408	0,000
	10	111446	111446	14,972	346,241	1692,485	0,000
	20	139178	139178	5,598	352,813	2036,598	0,000
	30	139524	139524	7,644	364,883	3294,308	0,000
1000	5	199778	199778	9,157	4214,491	7788,846	0,000
	10	219433	219433	12,694	4129,344	5566,288	0,000
	20	226480	226480	21,455	4193,865	1875,73	0,000
	30	226548	226536	33,173	4153,875	8273,707	-0,005
2500	5	555190	555190	53,44	*	4490,332	0,000
	10	548500	548500	73,297	*	16484,682	0,000
	20	502534	501682	130,122	*	9840,365	-0,170
	30	555243	555124	195,778	*	9040,988	-0,021
5000	5	1003022	1003022	208,084	*	14521,917	0,000
	10	1006494	1006494	291,814	*	8598,139	0,000
	20	1007790	1006458	534,504	*	12743,39	-0,132
	30	1012066	1012004	797,609	*	4708,615	-0,006
10000	5	2231298	2231298	839,331	*	13334,62	0,000
	10	2012279	2012279	1497,29	*	19615,044	0,000
	20	2017828	2005891	3465,757	*	27427,227	-0,595
	30	2015860	2015818	5266,108	*	13829,13	-0,002

Note: * means the instance uses more than 1 hour or uses up memory.

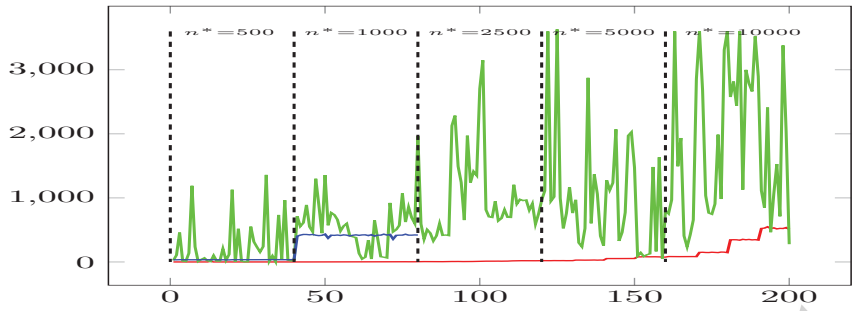

Figure 3: Solution time of DPI algorithm (—) compared to CPLEX 12.5 (—) and *DPB* algorithm (—)

Table 2: Comparative performance of *DPI* and CPLEX 12.5 with weakly correlated instances

N	DPI_{cpu} (s)	$CPLEX_{cpu}$ (s)	NB_{opt}
5	546	11198	0
10	916	12522	0
20	4257	16029	7
30	14317	34323	8

4. Conclusion

In this study, we consider the Knapsack Problem with setup. This problem can be used to model a wide range of concrete industrial problems, including order acceptance and production scheduling. We designed a dynamic programming algorithm using a new space reduction technique that converts a KPS solution into an integer index. This technique allows us to considerably reduce the storage requirement from $O(b \times n^*)$ to $O(b)$. The results show the promising impact of this technique on the performance of our algorithm, particularly in terms of solving large instances in a short computation time.

The use of the space reduction technique is specific to KPS particularly because of its special structure which allows to decompose the problem into sub-problems. Further studies, some of which are currently underway in our

laboratory, are nedded to extend the use of the space reduction technique for an other general problem like the Multiple Knapsack Problem with Setup (MKPS).

[1] S. Martello, P. Toth, Knapsack problems: Algorithms and computer implementations, John Wiley & Sons, 605 Third Avenue, New York, NY 10158-0012, USA, 1990.

[2] H. Kellerer, U. Pferschy, D. Pisinger, Knapsack problem, Springer, Verlag Berlin Heidelberg, 2004.

[3] E. Chajakis, M. Guignard, Exact algorithms for the setup knapsack problem, INFOR 32 (1994) 124–142.

[4] S. Michel, N. Perrot, F. Vanderbeck, Knapsack problems with setups, European Journal of Operational Research 196 (3) (2009) 909–918.

[5] E. Horowitz, S. Sahni, Computing partitions with applications to the knapsack problem, Journal of ACM 21 (1974) 277–292.

[6] U. Akinc, approximate and exact algorithms for the fixed-charge knapsack problem, European Journal of Operational Research 170 (2004) 363–375.

[7] L. McLay, S. H. Jacobson, Algorithms for the bounded set-up knapsack problem, Discrete Optimization 4 (2007) 206–212.

[8] H. Sural, L. Wassenhove, C. Potts, The Bounded Knapsack Problem with Setups, INSEAD working paper, INSEAD, 1997.

[9] L. McLay, S. H. Jacobson, Integer knapsack problems with set-up weights., Comp. Opt. and Appl. 37 (1) (2007) 35–47.

[10] L. McLay, Designing Aviation Security Systems: Theory and Practice, University of Illinois at Urbana-Champaign, 2006.

[11] L. McLay, S. H. Jacobson, J. E. Kobza, A multilevel passenger screening problem for aviation security., Naval Research Logistics 53 (3) (2006) 183–197.

[12] R. Bellman, Dynamic Programming, 1st Edition, Princeton University Press, Princeton, NJ, USA, 1957.

[13] P. Toth, Dynamic programming algorithms for the zero-one knapsack problem, computing 25 (1980) 29–45.

[14] D. Pisinger, A minimal algorithm for the 0-1 knapsack problem., Operations Research 45 (1994) 758–767.

[15] G. Plateau, M. Elkihel, A hybrid method for the 0-1 knapsack problem, Methods of Operations Research 49 (1985) 277–293.

[16] Y. Yang, Knapsack problems with setup, Dissertation, Auburn university (August 7 2006).

Appendix A. Detailed algorithms

In this appendix, we provide the detailed algorithms presented in this paper.

Algorithm 3 SolutionToIndex

Data: N, S
Result: $index$
$k \leftarrow 0$
$index \leftarrow 0$
for $x \leftarrow 1$ to N **do**
  if $x \in S$ **then**
    $k \leftarrow k + 1$
  else
    $index \leftarrow index + Binomial(x - 1, k - 1)$
  end if
end for
End Algorithm

Algorithm 1 DP_{basic}

Data: Problem KPS

Result: $V_b^{n^*}$,Solution S_{opt}

for $\beta \leftarrow 0$ to b **do**

if $\beta < a_{t(1),1} + d_{t(1)}$ **then**

$(V_\beta^1)^1 \leftarrow -\infty$

else

$(V_\beta^1)^1 \leftarrow c_{t(1),1} + f_1$

$(S_\beta^1)^1[1] \leftarrow 1$

end if

$(V_\beta^1)^0 \leftarrow 0$

$(S_\beta^1)^0[1] \leftarrow 0$

end for

for $k \leftarrow 2$ to n^* **do**

$\alpha \leftarrow a_{t(k),n(k)}$

$\delta \leftarrow d_{t(k)}$

if $t(k) = t(k-1)$ **then**

for $\beta \leftarrow 0$ to b **do**

if $\beta < \alpha + \delta$ **then**

$(V_\beta^k)^1 \leftarrow v(k-1, \beta)^1$

$(S_\beta^k)^1 \leftarrow (S_\beta^{k-1})^1$

$(S_\beta^k)^1[k] \leftarrow 0$

Algorithm 1 DP_{basic}(continued)

else

$(V_\beta^k)^1 \leftarrow \mathbf{max\{}(V_\beta^{k-1})^1;$

$(V_{\beta-\alpha}^{k-1})^1 + c_{t(k),n(k)};$

$(V_{\beta-\alpha-\delta}^{k-1})^0 + c_{t(k),n(k)} + f_{t(k)}\mathbf{\}}$

if $(V_\beta^k)^1 = (V_\beta^{k-1})^1$ **then**

$(S_\beta^k)^1 \leftarrow (S_\beta^{k-1})^1$

$(S_\beta^k)^1[k] \leftarrow 0$

else

if $(V_\beta^k)^1 = (V_{\beta-\alpha}^{k-1})^1 + c_{t(k),n(k)}$ **then**

$(S_\beta^k)^1 \leftarrow (S_{\beta-\alpha}^{k-1})^1$

else

$(S_\beta^k)^1 \leftarrow (S_{\beta-\alpha-\delta}^{k-1})^0$

end if

$(S_\beta^k)^1[k] \leftarrow 1$

end if

end if

$(V_\beta^k)^0 \leftarrow (V_\beta^{k-1})^0$

$(S_\beta^k)^0 \leftarrow (S_\beta^{k-1})^0$

$(S_\beta^k)^0[k] \leftarrow 0$

end for

else

for $\beta \leftarrow 0$ to b **do**

if $\beta < \alpha + \delta$ **then**

$(V_\beta^k)^1 \leftarrow -\infty$

else

$(V_\beta^k)^1 \leftarrow \mathbf{max\{}$

$(V_{\beta-\alpha-\delta}^{k-1})^1 + c_{t(k),n(k)} + f_{t(k)};$

$(V_{\beta-\alpha-\delta}^{k-1})^0 + c_{t(k),n(k)} + f_{t(k)}\mathbf{\}}$

Algorithm 1 DP_{basic} (continued)

if $(V_\beta^k)^1 = (V_{\beta-\alpha-\delta}^{k-1})^1 + c_{t(k),n(k)} + f_{t(k)}$ **then**
$(S_\beta^k)^1 \leftarrow (S_{\beta-\alpha-\delta}^{k-1})^1$
else
$(S_\beta^k)^1 \leftarrow (S_{\beta-\alpha-\delta}^{k-1})^0$
end if
$(S_\beta^k)^1[k] \leftarrow 1$
end if
$(V_\beta^k)^0 \leftarrow$ **max{** $(V_\beta^{k-1})^1; (V_\beta^{k-1})^0$**}**
if $(V_\beta^k)^0 = (V_\beta^{k-1})^1$ **then**
$(S_\beta^k)^0 \leftarrow (S_\beta^{k-1})^1$
else
$(S_\beta^k)^0 \leftarrow (S_\beta^{k-1})^0$
end if
end for
end if
end for
if $(V_b^{n^*})^1 > (V_b^{n^*})^0$ **then**
$V_b^{n^*} \leftarrow (V_b^{n^*})^1$
$S_{opt} \leftarrow (S_b^{n^*})^1$
else
$V_b^{n^*} \leftarrow (V_b^{n^*})^0$
$S_{opt} \leftarrow (S_b^{n^*})^0$
end if
End Algorithm

Algorithm 2 Binomial

Data: N, k

Result: b

if $(N \leq 0 || k \leq 0)$ **then**

$b \leftarrow 0$

else

$b \leftarrow 1$

for $i \leftarrow 0$ to $k - 1$ **do**

$b \leftarrow b \times (N - i)$

$b \leftarrow b \div (i + 1)$

end for

end if

End Algorithm

Algorithm 4 IndexToSolution

Data: $N, k, index$

Result: S

if $index < 0 || N \leq 0 || k \leq 0$ **then**

break

end if

$b \leftarrow Binomial(N - 1, k - 1)$

if $index < b$ **then**

$IndexToSolution(N - 1, k - 1, index)$

ADD N to S

else

$IndexToSolution(N - 1, k, index - b)$

end if

End Algorithm

Algorithm 5 Improved DP procedure

Data: Problem KPS

Result: $V_b^{n^*}$,Solution S_{opt}

for $\beta \leftarrow 0$ to b **do**

if $\beta < a_{t(1),1} + d_{t(1)}$ **then**

$(V_\beta^1)^1 \leftarrow -\infty$

$(nbCL_\beta^1)^1 \leftarrow 0$

$(index_\beta^1)^1 \leftarrow SolutionToIndex(N, \oslash)$

else

$(V_\beta^1)^1 \leftarrow c_{t(1),1} + f_1$

$ADD\ t(1)\ to\ S$

$(nbCL_\beta^1)^1 \leftarrow 1$

$(index_\beta^1)^1 \leftarrow SolutionToIndex(N, S)$

end if

$(V_\beta^1)^0 \leftarrow 0$

$(nbCL_\beta^1)^0 \leftarrow 0$

$(index_\beta^1)^0 \leftarrow SolutionToIndex(N, \oslash)$

end for

for $k \leftarrow 2$ to n^* **do**

$\alpha \leftarrow a_{t(k),n(k)}$

$\delta \leftarrow d_{t(k)}$

if $t(k) = t(k-1)$ **then**

for $\beta \leftarrow 0$ to b **do**

if $\beta < \alpha + \delta$ **then**

$(V_\beta^k)^1 \leftarrow v(k-1, \beta)^1$

$(nbCL_\beta^k)^1 \leftarrow (nbCL_\beta^{k-1})^1$

$(index_\beta^k)^1 \leftarrow (index_\beta^{k-1})^1$

else

$(V_\beta^k)^1 \leftarrow \mathbf{max}\{(V_\beta^{k-1})^1;$

$(V_{\beta-\alpha}^{k-1})^1 + c_{t(k),n(k)};$

$(V_{\beta-\alpha-\delta}^{k-1})^0 + c_{t(k),n(k)} + f_{t(k)}\}$

Algorithm 5 Improved DP procedure(continued)

if $(V_\beta^k)^1 = (V_\beta^{k-1})^1$ **then**
$(nbCL_\beta^k)^1 \leftarrow (nbCL_\beta^{k-1})^1$
$(index_\beta^k)^1 \leftarrow (index_\beta^{k-1})^1$
else
if $(V_\beta^k)^1 = (V_{\beta-\alpha}^{k-1})^1 + c_{t(k),n(k)}$ **then**
$S \leftarrow IndexToSolution(N, (nbCL_{\beta-\alpha}^{k-1})^1, (index_{\beta-\alpha}^{k-1})^1)$
$ADD\ t(k)\ to\ S$
$(nbCL_\beta^k)^1 \leftarrow |S|$
$(index_\beta^k)^1 \leftarrow SolutionToIndex(N, S)$
else
$S \leftarrow IndexToSolution(N, (nbCL_{\beta-\alpha-\delta}^{k-1})^0, (index_{\beta-\alpha-\delta}^{k-1})^0)$
$ADD\ t(k)\ to\ S$
$(nbCL_\beta^k)^1 \leftarrow |S|$
$(index_\beta^k)^1 \leftarrow SolutionToIndex(N, S)$
end if
end if
end if
$(V_\beta^k)^0 \leftarrow (V_\beta^{k-1})^0$
$(nbCL_\beta^k)^0 \leftarrow (nbCL_\beta^{k-1})^0$
$(index_\beta^k)^0 \leftarrow (index_\beta^{k-1})^0$
end for
else
for $\beta \leftarrow 0$ to b **do**
if $\beta < \alpha + \delta$ **then**
$(V_\beta^k)^1 \leftarrow -\infty$
$(nbCL_\beta^1)^1 \leftarrow 0$
$(index_\beta^1)^1 \leftarrow SolutionToIndex(N, \oslash)$
else
$(V_\beta^k)^1 \leftarrow \mathbf{max\{}$
$(V_{\beta-\alpha-\delta}^{k-1})^1 + c_{t(k),n(k)} + f_{t(k)};$
$(V_{\beta-\alpha-\delta}^{k-1})^0 + c_{t(k),n(k)} + f_{t(k)}\mathbf{\}}$

Algorithm 5 Improved DP procedure (continued)

if $(V_\beta^k)^1 = (V_{\beta-\alpha-\delta}^{k-1})^1 + c_{t(k),n(k)} + f_{t(k)}$ **then**
$S \leftarrow IndexToSolution(N, (nbCL_{\beta-\alpha-\delta}^{k-1})^1, (index_{\beta-\alpha-\delta}^{k-1})^1)$
$ADD\ t(k)\ to\ S$
$(nbCL_\beta^k)^1 \leftarrow |S|$
$(index_\beta^k)^1 \leftarrow SolutionToIndex(N, S)$
else
$S \leftarrow IndexToSolution(N, (nbCL_{\beta-\alpha-\delta}^{k-1})^0, (index_{\beta-\alpha-\delta}^{k-1})^0)$
$ADD\ t(k)\ to\ S$
$(nbCL_\beta^k)^1 \leftarrow |S|$
$(index_\beta^k)^1 \leftarrow SolutionToIndex(N, S)$
end if
end if
$(V_\beta^k)^0 \leftarrow \mathbf{max\{}\ (V_\beta^{k-1})^1; (V_\beta^{k-1})^0\mathbf{\}}$
if $(V_\beta^k)^0 = (V_\beta^{k-1})^1$ **then**
$(nbCL_\beta^k)^0 \leftarrow (nbCL_\beta^{k-1})^1$
$(index_\beta^k)^0 \leftarrow (index_\beta^{k-1})^1$
else
$(nbCL_\beta^k)^0 \leftarrow (nbCL_\beta^{k-1})^0$
$(index_\beta^k)^0 \leftarrow (index_\beta^{k-1})^0$
end if
end for
end if
end for
if $(V_b^{n^*})^1 > (V_b^{n^*})^0$ **then**
$V_b^{n^*} \leftarrow (V_b^{n^*})^1$
$S \leftarrow IndexToSolution(N, (nbCL_b^{n^*})^1, (index_b^{n^*})^1)$
else
$V_b^{n^*} \leftarrow (V_b^{n^*})^0$
$S \leftarrow IndexToSolution(N, (nbCL_b^{n^*})^1, (index_b^{n^*})^1)$
end if
$S_{opt} \leftarrow$ optimal solution of KPS[S] using CPLEX.
End Algorithm

Appendix B. Detailed computational results

In this appendix, we provide the detailed results of our computational experiments. In table B.3, the following notations are considered:

- n^*: the total number of items.
- N: the number of classes.
- *problem* : The number of the problem.
- $CPLEX_{stat}$:
  - 101: Optimal integer solution found
  - 107: Time limit exceeded, integer solution exists
  - 109: Error termination, integer solution exists
- $CPLEX_{obj}$: value provided by CPLEX.
- DPI_{obj}: value provided by the DPI algorithm.
- $CPLEX_{cpu}$: the running time of CPLEX.
- DPI_{cpu}: the running time of the DPI algorithm.
- $GAP(\%) = \frac{100\times(CPLEX_{obj}-DPI_{obj})}{CPLEX_{obj}}$

Table B.3: *Detailed comparison between DP and CPLEX*

n^*	N	*problem*	$CPLEX_{stat}$	$CPLEX_{obj}$	DPI_{obj}	$CPLEX_{cpu}$ (s)	DPI_{cpu} (s)	$GAP(\%)$
500	5	1	101	10965	10965	25,338	0,353	0,000
		2	101	10964	10964	107,449	0,474	0,000
		3	101	11155	11155	459,015	0,218	0,000
		4	101	11061	11061	14,354	0,259	0,000
		5	101	11478	11478	7,474	0,223	0,000

continued on next page

TabB.3: *continued...*

n_{tot}	N	*problem*	$CPLEX_{stat}$	$CPLEX_{obj}$	DPI_{obj}	$CPLEX_{cpu}$ (s)	DPI_{cpu} (s)	$GAP(\%)$
		6	101	11244	11244	137,081	0,215	0,000
		7	109	10881	10881	1189,863	0,213	0,000
		8	101	11049	11049	237,249	0,419	0,000
		9	101	10971	10971	8,358	0,21	0,000
		10	101	10970	10970	39,227	0,489	0,000
	10	1	101	11202	11202	63,127	0,35	0,000
		2	101	10929	10929	17,795	0,484	0,000
		3	101	11214	11214	7,017	11,317	0,000
		4	101	10742	10742	100,328	0,271	0,000
		5	101	11041	11041	13,442	0,316	0,000
		6	101	11851	11851	12,496	0,363	0,000
		7	101	11385	11385	205,392	0,301	0,000
		8	101	11027	11027	26,681	0,621	0,000
		9	101	10873	10873	117,878	0,662	0,000
		10	101	11182	11182	1128,329	0,287	0,000
	20	1	101	14004	14004	1,596	0,484	0,000
		2	109	14018	14018	518,419	0,499	0,000
		3	101	13872	13872	13,2	0,574	0,000
		4	101	13809	13809	1,898	0,498	0,000
		5	101	14211	14211	3,697	0,62	0,000
		6	109	13952	13952	551,919	0,509	0,000
		7	101	13372	13372	315,968	0,646	0,000
		8	101	13589	13589	232,986	0,484	0,000
		9	101	14824	14824	156,877	0,779	0,000
		10	101	13527	13527	240,038	0,505	0,000
	30	1	101	14284	14284	1359,757	0,733	0,000
		2	101	14101	14101	13,36	0,744	0,000
		3	101	13673	13673	94,007	0,767	0,000
		4	101	13781	13781	8,401	0,718	0,000

continued on next page

TabB.3: *continued...*

n_{tot}	N	*problem*	$CPLEX_{stat}$	$CPLEX_{obj}$	DPI_{obj}	$CPLEX_{cpu}$ (s)	DPI_{cpu} (s)	$GAP(\%)$
		5	109	14034	14034	734,542	0,772	0,000
		6	101	14019	14019	47,291	0,785	0,000
		7	101	14384	14384	965,045	0,887	0,000
		8	101	13746	13746	28,503	0,72	0,000
		9	101	13683	13683	16,076	0,774	0,000
		10	101	13819	13819	27,326	0,744	0,000
1000	5	1	109	19351	19351	714,753	0,963	0,000
		2	109	19815	19815	556,295	1,004	0,000
		3	109	20091	20091	611,398	0,836	0,000
		4	101	20239	20239	890,992	1,059	0,000
		5	109	20181	20181	548,15	0,989	0,000
		6	109	19753	19753	471,706	1,012	0,000
		7	101	19951	19951	1299,87	0,794	0,000
		8	101	19793	19793	888,416	0,82	0,000
		9	109	19715	19715	449,741	0,81	0,000
		10	101	20889	20889	1357,525	0,87	0,000
	10	1	109	21568	21568	572,755	1,214	0,000
		2	109	22107	22107	768,963	1,267	0,000
		3	109	22078	22078	737,018	1,285	0,000
		4	109	22106	22106	660,8	1,206	0,000
		5	109	21481	21481	483,529	1,134	0,000
		6	109	22388	22388	560,673	1,366	0,000
		7	109	22305	22305	599,859	1,382	0,000
		8	109	21373	21373	408,312	1,303	0,000
		9	109	22290	22290	393,366	1,304	0,000
		10	109	21737	21737	381,013	1,233	0,000
	20	1	101	22590	22590	132,661	2,108	0,000
		2	101	22488	22488	46,319	2,042	0,000
		3	101	22760	22760	74,289	2,23	0,000

continued on next page

TabB.3: *continued...*

n_{tot}	N	*problem*	$CPLEX_{stat}$	$CPLEX_{obj}$	DPI_{obj}	$CPLEX_{cpu}$ (s)	DPI_{cpu} (s)	$GAP(\%)$
		4	101	22464	22464	341,027	2,167	0,000
		5	101	22787	22787	45,621	2,063	0,000
		6	101	23157	23157	647,894	2,63	0,000
		7	101	22138	22138	368,495	2,043	0,000
		8	101	22527	22527	82,216	2,059	0,000
		9	101	22888	22888	74,321	2,05	0,000
		10	101	22681	22681	62,887	2,063	0,000
	30	1	101	22141	22141	922,349	3,083	0,000
		2	101	22116	22116	470,203	3,089	0,000
		3	101	22641	22641	503,956	3,336	0,000
		4	109	22723	22723	572,125	3,541	0,000
		5	101	22635	22635	1069,48	3,394	0,000
		6	101	23167	23167	622,835	3,626	0,000
		7	101	22489	22489	855,863	3,089	0,000
		8	101	22588	22588	691,021	3,192	0,000
		9	**109**	**22758**	**22770**	**600,828**	**3,192**	**-0,053**
		10	101	23278	23278	1965,047	3,631	0,000
2500	5	1	109	55207	55207	593,926	5,403	0,000
		2	109	55719	55719	356,091	5,706	0,000
		3	109	55811	55811	504,111	5,396	0,000
		4	109	55593	55593	451,606	5,182	0,000
		5	109	56653	56653	329,16	5,517	0,000
		6	109	54612	54612	407,961	5,03	0,000
		7	109	54739	54739	598,766	5,243	0,000
		8	109	55407	55407	415,84	5,329	0,000
		9	109	55746	55746	421,006	5,378	0,000
		10	109	55703	55703	411,865	5,256	0,000
	10	1	109	55021	55021	2128,035	7,227	0,000
		2	109	54143	54143	2285,13	7,136	0,000

continued on next page

TabB.3: *continued...*

n_{tot}	N	*problem*	$CPLEX_{stat}$	$CPLEX_{obj}$	DPI_{obj}	$CPLEX_{cpu}$ (s)	DPI_{cpu} (s)	$GAP(\%)$
		3	109	54359	54359	1493,945	7,72	0,000
		4	109	55262	55262	1241,91	7,367	0,000
		5	109	55044	55044	653,988	7,425	0,000
		6	109	54566	54566	1972,014	7,248	0,000
		7	109	55212	55212	1241,075	7,351	0,000
		8	109	55033	55033	1411,667	7,313	0,000
		9	109	54712	54712	1351,457	7,251	0,000
		10	101	55148	55148	2705,461	7,259	0,000
	20	1	109	50567	50567	3149,109	13,585	0,000
		2	**109**	**49969**	**50051**	**814,164**	**12,571**	**-0,164**
		3	**109**	**49659**	**49913**	**704,048**	**13,053**	**-0,511**
		4	**109**	**49753**	**49830**	**695,187**	**12,774**	**-0,155**
		5	109	49735	49735	947,101	12,537	0,000
		6	**109**	**49541**	**49695**	**642,253**	**13,135**	**-0,311**
		7	**109**	**49331**	**49446**	**680,97**	**12,227**	**-0,233**
		8	109	50167	50167	814,577	13,107	0,000
		9	**109**	**50897**	**51067**	**697,585**	**13,458**	**-0,334**
		10	109	52063	52063	695,371	13,675	0,000
	30	1	109	49891	49891	1205,896	19,256	0,000
		2	**109**	**56047**	**56128**	**907,683**	**19,576**	**-0,145**
		3	109	55701	55701	977,224	19,236	0,000
		4	**109**	**56185**	**56200**	**958,493**	**19,967**	**-0,027**
		5	109	56211	56211	967,385	19,217	0,000
		6	**109**	**56873**	**56894**	**809,311**	**19,502**	**-0,037**
		7	109	55702	55702	909,367	19,043	0,000
		8	109	55670	55670	590,808	20,197	0,000
		9	109	56933	56933	750,894	19,941	0,000
		10	**109**	**55911**	**55913**	**963,927**	**19,843**	**-0,004**
5000	5	1	109	99661	99661	1109,984	20,863	0,000

continued on next page

TabB.3: *continued...*

n_{tot}	N	*problem*	$CPLEX_{stat}$	$CPLEX_{obj}$	DPI_{obj}	$CPLEX_{cpu}$ (s)	DPI_{cpu} (s)	$GAP(\%)$
		2	107	100371	100371	3600,299	20,376	0,000
		3	109	99253	99253	950,988	20,31	0,000
		4	109	100709	100709	1016,152	21,523	0,000
		5	107	101143	101143	3631,252	21,05	0,000
		6	109	100015	100015	995,998	20,82	0,000
		7	109	100315	100315	525,115	21,419	0,000
		8	109	100189	100189	759,107	20,308	0,000
		9	109	100499	100499	1169,966	20,609	0,000
		10	109	100867	100867	763,056	20,806	0,000
	10	1	109	100447	100447	970,453	29,022	0,000
		2	109	101343	101343	298,755	29,485	0,000
		3	101	99553	99553	238,593	28,999	0,000
		4	109	100389	100389	525,091	28,412	0,000
		5	109	99615	99615	2875,407	28,828	0,000
		6	109	100905	100905	603,375	29,441	0,000
		7	101	101507	101507	1367,232	30,519	0,000
		8	109	100989	100989	1059,983	28,299	0,000
		9	109	101421	101421	403,823	30,142	0,000
		10	109	100325	100325	255,427	28,667	0,000
	20	1	**109**	**101133**	**101585**	**1277,587**	**53,224**	**-0,447**
		2	109	99785	99785	1122,562	52,479	0,000
		3	**109**	**99801**	**99889**	**1233,746**	**53,944**	**-0,088**
		4	101	100847	100847	2071,302	51,824	0,000
		5	101	100661	100661	327,811	53,213	0,000
		6	109	101023	101023	461,694	54,994	0,000
		7	109	100811	100811	773,696	54,238	0,000
		8	**109**	**100019**	**100133**	**1973,947**	**52,621**	**-0,114**
		9	**109**	**100959**	**101283**	**2026,137**	**53,786**	**-0,321**
		10	**109**	**101419**	**101773**	**1474,908**	**54,181**	**-0,349**

continued on next page

TabB.3: *continued...*

n_{tot}	N	*problem*	$CPLEX_{stat}$	$CPLEX_{obj}$	DPI_{obj}	$CPLEX_{cpu}$ (s)	DPI_{cpu} (s)	$GAP(\%)$
	30	1	101	101929	101929	94,855	80,596	0,000
		2	101	101839	101839	145,199	79,341	0,000
		3	101	100244	100244	86,705	83,42	0,000
		4	101	100247	100247	112,508	80,319	0,000
		5	101	100613	100613	127,165	81,066	0,000
		6	**109**	**101367**	**101429**	**1483,399**	**80,826**	**-0,061**
		7	101	101963	101963	165,268	81,543	0,000
		8	101	101525	101525	1635,147	78,723	0,000
		9	101	101406	101406	54,244	75,484	0,000
		10	109	100871	100871	804,125	76,291	0,000
10000	5	1	109	222561	222561	755,381	84,652	0,000
		2	109	223710	223710	967,197	85,187	0,000
		3	107	223263	223263	3600,394	82,428	0,000
		4	109	223212	223212	1494,032	84,199	0,000
		5	109	224012	224012	1935,735	82,632	0,000
		6	109	222838	222838	407,417	84,652	0,000
		7	109	222511	222511	236,333	84,128	0,000
		8	109	221362	221362	423,078	84,134	0,000
		9	109	223404	223404	655,955	84,257	0,000
		10	109	224425	224425	2859,098	83,062	0,000
	10	1	107	203813	203813	3601,15	149,627	0,000
		2	109	201393	201393	2679,776	151,595	0,000
		3	109	200229	200229	1036,392	147,919	0,000
		4	109	198905	198905	764,042	147,104	0,000
		5	109	201261	201261	745,574	152,11	0,000
		6	109	200848	200848	905,471	149,147	0,000
		7	109	201297	201297	1988,604	146,336	0,000
		8	109	201046	201046	982,906	154,605	0,000
		9	109	202163	202163	3310,912	153,322	0,000

continued on next page

TabB.3: *continued...*

n_{tot}	N	*problem*	$CPLEX_{stat}$	$CPLEX_{obj}$	DPI_{obj}	$CPLEX_{cpu}$ (s)	DPI_{cpu} (s)	$GAP(\%)$
		10	107	201324	201324	3600,217	145,525	0,000
	20	1	**109**	**201654**	**203934**	**2575,263**	**345,869**	**-1,131**
		2	**109**	**199809**	**201679**	**2816,922**	**347,333**	**-0,936**
		3	**109**	**200420**	**200920**	**2438,595**	**346,315**	**-0,249**
		4	107	200500	200500	3600,303	339,765	0,000
		5	109	202400	202400	1129,033	353,676	0,000
		6	**109**	**199629**	**200715**	**2999,638**	**346,687**	**-0,544**
		7	**109**	**201386**	**202286**	**2787,459**	**351,743**	**-0,447**
		8	**109**	**200005**	**202291**	**2663,111**	**344,86**	**-1,143**
		9	**109**	**200291**	**201818**	**3522,688**	**348,292**	**-0,762**
		10	**109**	**199797**	**201285**	**2894,215**	**341,217**	**-0,745**
	30	1	109	201438	201438	1114,503	523,28	0,000
		2	101	201466	201466	848,357	528,609	0,000
		3	**109**	**201995**	**202037**	**2410,888**	**549,032**	**-0,021**
		4	101	200886	200886	464,622	522,142	0,000
		5	101	202139	202139	1056,574	532,605	0,000
		6	101	201864	201864	1529,494	516,298	0,000
		7	101	202001	202001	712,613	524,321	0,000
		8	101	202196	202196	3381,738	522,445	0,000
		9	101	201980	201980	2032,745	537,225	0,000
		10	101	199853	199853	277,596	510,151	0,000