Computational Optimization and Applications, 23, 27–45, 2002
© 2002 Kluwer Academic Publishers. Manufactured in The Netherlands.

An Efficient Algorithm for the Knapsack Sharing Problem

MHAND HIFI hifi@laria.u-picardie.fr
LaRIA, Université de Picardie Jules Verne, 5 rue du Movlin Neuf, 80000 Amiens, France

SLIM SADFI sadfi@univ-paris1.fr
ABDELKADER SBIHI sbihi@univ-paris1.fr
CERMSEM-CNRS URA 8095, Maison des Sciences Economiques, Université de Paris 1 Panthéon-Sorbonne, 106-112 Boulevard de l'Hôpital, 75647 Paris Cedex 13, France

Abstract. The Knapsack Sharing Problem (KSP) is an NP-Hard combinatorial optimization problem, admitted in numerous real world applications. In the KSP, we have a knapsack of capacity c and a set of n objects, namely $\mathcal{N}$, where each object j, $j = 1, \ldots, n$, is associated with a profit p_j and a weight w_j. The set of objects $\mathcal{N}$ is composed of *m different classes* of objects J_i, $i = 1, \ldots, m$, and $\mathcal{N} = \bigcup_{i=1}^{m} J_i$. The aim is to determine a subset of objects to be included in the knapsack that realizes a max-min value over all classes.

In this article, we solve the KSP using an approximate solution method based upon tabu search. First, we describe a simple local search in which a depth parameter and a tabu list are used. Next, we enhance the algorithm by introducing some intensifying and diversifying strategies. The two versions of the algorithm yield satisfactory results within reasonable computational time. Extensive computational testing on problem instances taken from the literature shows the effectiveness of the proposed approach.

Keywords: combinatorial optimization, heuristic, knapsack, local search

1. Introduction

Unless P = NP, many interesting combinatorial optimization problems cannot be solved exactly within a reasonable amount of time. Consequently, heuristics must be used to solve large real world problems. Approximate algorithms may be divided into two main classes: *general purpose algorithms* designed independently from the optimization problem at hand, and *tailored algorithms* specifically designed for a given problem.

In this article we propose a *general approximate algorithm* for solving the Knapsack Sharing Problem (KSP). This problem has a wide range of commercial applications (see Brown [1] and Tang [17]). In the KSP, we have a set of n objects represented by $\mathcal{N} = \{1, \ldots, j, \ldots, n\}$, where each object j is associated with a profit p_j and a weight w_j. Furthermore, the set $\mathcal{N}(= J_1 \cup J_2 \cup \cdots \cup J_n)$ is composed of *m different classes of objects*, i.e., for each pair (p, q), $p \neq q$, $p \leq m$, and $q \leq m$, we have $J_p \cap J_q = \emptyset$ and $\bigcup_{i=1}^{m} J_i = \mathcal{N}$. Moreover, given a knapsack of capacity c, we wish to determine a subset of objects to be included in the knapsack. The KSP is equivalent to maximizing the minimal value of a set of linear functions subject to single linear constraint. Indeed, if we define $x_j = 1$ if the object j is in the solution set, and $x_j = 0$ otherwise, the mathematical formulation of the

problem can be stated as follows:

$$(KSP)\begin{cases} \text{Max} & \min_{1\le i\le m}\left\{\sum_{j\in J_i} p_j x_j\right\} \\ \text{Subject to} & \sum_{j\in\mathcal{N}} w_j x_j \le c \\ & x_j \in \{0,1\}, \quad \text{for } j = 1,\ldots,n \end{cases}$$

We may assume without loss of generality that w_j, p_j (for $j = 1, \ldots, n$), and c are positive integers and that $\sum_{j\in\mathcal{N}} w_j > c$. In what follows, we consider that the elements of each class J_i, $i = 1, \ldots, m$, are indexed in decreasing order such that $p_1/w_1 \ge p_2/w_2 \ge \cdots \ge p_{|J_i|}/w_{|J_i|}$. Notice that if we have two terms that realize the same coefficient, then we first accept the term with the greater profit. The problem so defined is NP hard, since it is a generalization of the famous single-constraint knapsack, i.e., when $m = 1$. The KSP is classified as *KSP*$(Bn/m/1)$ (see Yamada and Futakawa [18]), which means that we have n objects of binary (B) type divided into m classes with one constraint.

In this article, we develop an approximate algorithm for the KSP. First, we propose a *Single-depth Tabu Search* (called STS) that combines a depth parameter with a tabu list. Next, we enhance the STS approach by introducing some intensifying and diversifying strategies: the resulting algorithm is called MTS (*Multiple-depth Tabu Search*). The rest of the article is organized as follows. First, in Section 2, we briefly review the literature concerning some sequential exact and approximate algorithms for the (general) knapsack problem. In Section 3, we describe the principle of the Tabu Search (TS). In Section 4, we present the TS procedure adapted for the KSP. We start (Section 4.1) by describing the solution representation and fitness function associated with an instance of the problem. In Section 4.2, a greedy procedure is proposed in order to produce an initial feasible solution. The main steps of the proposed algorithm are detailed in Sections 4.3–4.6. In Sections 4.3 and 4.4, we describe the main principle of the first version of the algorithm, called *Single-depth Tabu Search*, and in Sections 4.5 and 4.6 we present the enhanced algorithm, called *Multiple-depth Tabu Search*. Finally, in Section 5, the performance of the proposed approach is tested on a set of problem instances extracted from the literature that have different sizes and densities.

2. Related work and knapsack problems

Because of its importance and despite its NP hardness, the *knapsack problem* has been widely studied in the literature. The design of the solution approaches depends in general on the particular framework of the application, which gives rise to the particular knapsack problem, and on the available computational resources.

For the (un)bounded *single constraint knapsack problem* (see Martello and Toth [13], Pisinger [15], and Syslo [16]), a large variety of solution methods have been devised. The problem has been solved optimally (and approximately) by dynamic programming, by the use of tree search procedures, and by other approaches. A good review of the

single-constraint knapsack problem and its associated approaches can be found in Gilmore and Gomory [5] and Martello and Toth [12].

Previous work has also developed several approaches for the general case, i.e., when the number of constraints is not limited to one. The problem is then called the *multidimensional* (or *multiconstraint*) *knapsack problem* (for a recent, detailed review of this problem, see Chu and Beasley [3]). When the number of constraints is limited to only two knapsack constraints, then the problem is referred to as the *particular bidimensional knapsack problem* (see Freville and Plateau [4]).

Another problem, namely the max-min allocation problem, has been studied by several authors (see Brown [2], Luss [11], Pang and Yu [14], and Tang [17]). Different exact and approximate approaches have been tailored especially for this problem. For the particular continuous KSP described by Kuno et al. [10], the authors have proposed a linear-time solution algorithm. Another algorithm has been proposed by Yamada and Futukawa ([18]). The KSP has been solved optimally by dynamic programming (see Hifi and Sadfi [9]) and by the use of tree search procedures (see Yamada et al. [19]).

3. The tabu search

3.1. The tabu search basics

Tabu Search (TS) has its roots in the field of artificial intelligence as well as in the field of optimization (see [6–8]). The heart of tabu search lies in its use of adaptive memory, which allows the search history to guide the solution process. In its simplest manifestations, adaptive memory is exploited to prohibit the search from reinvestigating solutions that have already been evaluated. However, the use of memory in our implementation is much more complex and calls upon memory functions that encourage search diversification and intensification. These memory components allow the search to escape from locally optimal solutions and in many cases to find optimal solutions.

One way of intelligently guiding a search process is to forbid (or discourage) certain solutions from being chosen based on information that suggests these solutions may duplicate, or significantly resemble, solutions encountered in the past. In TS, this is often done by defining suitable attributes of moves or solutions and then imposing restrictions on a set of the attributes, depending on the search history. Two prominent ways for exploiting search history in TS are through recency memory and frequency memory. Recency memory is typically (though not invariably) a short-term memory that is managed by structures or arrays called “tabu lists,” while frequency memory more usually fulfills a long-term search function. One standard form of recency memory discourages moves that lead to solutions with attributes shared by other solutions recently visited. A standard form of frequency memory, on the other hand, either discourages moves leading to solutions with attributes shared by other solutions visited during the search or, alternatively, encourages moves leading to solutions whose attributes have rarely been seen before. Another standard form of frequency memory is defined over subsets of elite solutions to fulfill an intensification function.

Short- and long-term components based on recency and frequency memory can be used separately or together in complementary TS strategies. Note that this approach operates

by implicitly modifying the neighborhood of the current solution. Tabu search in general includes many enhancements to the scheme sketched here.

3.2. TS foundations

TS begins in the same way as an ordinary local or neighborhood search, proceeding iteratively from one point (solution) to another until a chosen termination criterion is satisfied. Each solution $s \in S$ (S denotes the set of the feasible solutions) has an associated neighborhood $N(s) \in S$, and each solution s' is reached from s by an operation called *move*. As an initial point of departure, we may contrast TS with a simple ascent method, where the goal is to maximize $f(s)$ (or with a corresponding descent method, where the goal is to minimize $f(s)$). Such a method only permits moves to neighbor solutions that improve the current objective function value, and it terminates when no improving solutions can be found. The final solution s obtained by an ascent method (maximizing context) is called a *local optimum*, since it is at least as good or better than all the solutions in its neighborhood. The evident shortcoming of an ascent method is that such a local optimum in most cases will not be a global optimum.

3.3. Use of memory

The memory structures in our approach operate by referring to four principal dimensions: recency, frequency, quality, and influence. *Recency-based* and *frequency-based* memory complement each other and have important characteristics. The *quality* dimension refers to the ability to differentiate the respective merits of solutions visited during the search. In this context, memory can be used to identify elements that are common to good solutions or to the paths that lead to such solutions. The fourth dimension, *influence*, considers the impact of the choices made during the search, not only on quality but also on structures (in the sense that quality may be regarded as a special form of influence). The memory used in tabu search is both *explicit* and *attributive*. Explicit memory records complete solutions, typically consisting of elite solutions visited during the search. An extension of this memory records highly attractive but unexplored neighbors of elite solutions. The memorized elite solutions (or their neighbors) are used to expand the local search.

These four principles adopted here are general and apply to the extended TS approach. In our study, we made some adjustments in order to apply them to the KSP case.

4. The TS for the KSP

4.1. Solution representation and fitness function

In our approach, and before describing the method, we give a suitable representation scheme, i.e., a way to represent a solution of the KSP. The standard KSP 0-1 binary representation is an obvious choice for the KSP, since it represents the underlying 0-1 integer variables. In our representation, we use an n-bit binary vector S, where n is the number of variables and

class	⟶		S_1			S_2			...	S_m	
column/item	⟶	1	2	3	1	2	3	4	...	1	2
bit subvector	⟶	1	0	0	1	1	0	0	...	0	1

Figure 1. Binary representation of the KSP solution.

$S(j) = 0$ or 1. $S(j) = 0$ (resp. $S(j) = 1$) means that $x_j = 0$ (resp. $x_j = 1$) in the solution of the KSP. Figure 1 shows the vector representation of this solution.

Generally, the binary representation can introduce infeasibility in the resulting solution. Therefore, a number of standard approaches exist to deal with dissatisfaction and infeasibility constraints. We can

1. use a representation that automatically ensures the feasibility of the solution;
2. separate the fitness function into two terms, the first one containing a subset that realizes a feasible solution to the KSP and the second one representing the amount of infeasibility of the current solution; and
3. design a local search procedure guaranteeing transformation of any obtained infeasible solution into a feasible one.

The above points were intensively used in several ways when genetic and simulated annealing algorithms were applied to solve some particular optimization problems (see Chu and Beasley [3], Glover and Laguna [7], and Hansen [8]). Here, we design another process in order to construct a current feasible solution. The proposed process uses the so-called *critical element*. The critical element of a class is the one that separates the class into two parts: the *right-critical* and the *left-critical* areas. In our case, the solution is considered as follows:

1. Consider a portion of each subvector S_i, $i = 1, \ldots, m$, and suppose that $S_i(k)$, $k \leq |J_i|$, is a *critical element* of the i-th class. Fix all elements of the left-critical region (of each class) to "one" and consider all elements of the right-critical region to be "free."
2. The obtained solution represents a feasible solution to the KSP if all free elements are set to zero. We can see that the solution can also be improved by setting some elements of the right-critical region to one.

In our method, we apply this simple approach of using heuristic operators. We prefer this approach because a "good" penalty function is often difficult to determine. By restricting the proposed algorithm to search only the feasible region up to the solution space, we obtain the following fitness function based entirely on the objective function to be maximized:

$$f(S) = \min\left\{ \sum_{j\in S_1} p_j S(j), \ldots, \sum_{j\in S_m} p_j S(j) \right\},$$

where $S = (S_1, \ldots, S_m)$. Note here that we try to identify the better subset (class), which produces the higher value and therefore the better KSP solution as well.

4.2. An initial solution to the KSP: A Greedy Heuristic (GH)

The proposed heuristic is of the greedy type, called herein *GH*. The main steps of the *GH* procedure are described by the following steps:

`Initialization.`

1. Set the initial capacity to zero, i.e., $SumCap \leftarrow 0$; (cumulate total capacity)
2. Set $min \leftarrow 1$, where *min* denotes the index of the class realizing the minimum (sub)solution;
3. For each $i \in \{1, \dots, m\}$, set $j_i \leftarrow 1$, $P_i \leftarrow 0$ and $W_i \leftarrow 0$, where P_i (resp. W_i) is the cumulate profit (resp. weight) of items picked in the i-th class.

`Repeat`

If $SumCap + w_{j_{min}} \leq c$ then set $SumCap = SumCap + w_{j_{min}}$;
- Set $j_{min} \leftarrow j_{min} + 1$;
- Let *min* be an index realizing $\min_{1 \leq i \leq m} \{P_i\}$;

`Until` $j_{min} > |J_{min}|$.

The GH procedure builds iteratively a feasible solution. Indeed, the main loop `repeat` begins by constructing a partial feasible solution. The obtained solution is completed later, using the same steps, by setting some elements to 0 or 1. Initially, all elements of each class are ranged in decreasing order of the proportion *profit/weight*. In `step 3` of the loop repeat, a current index *min*, representing a favorite class, is selected. This current index represents the class such that the sum of its fixed-items profits in the subsolution is the smallest one. The first element, denoted j_{min} of the *min*-th class, is added to the current solution if (1) the element has not been selected before and (2) its weight does not exceed the residual capacity of the resulting knapsack. The process is iterated until there is no possibility of fixing any other element in the current class to put in the knapsack. It is easy to see that this solution is feasible.

4.3. The critical element and tabu list

In this section, we try to improve the solution produced by the procedure *GH*. We do so by applying a *neighborhood search* throughout some elements called *critical elements*. We recall that each class J_i has a critical element r_{J_i}, which denotes the index of a particular element of the class J_i that has a particular knapsack capacity, namely, $\bar{c}_{J_i}$. Let us consider the following problems by setting $\bar{c}_{J_i} \leq c, \forall i \in \{1, \dots, m\}$, where $\bar{c}_{J_i}$ is a nonnegative integer:

$$\left(SK_{J_i}^{\bar{c}_{J_i}}\right) \begin{cases} \text{Max} & \sum_{j \in J_i} p_j x_j \\ \text{subject to} & \sum_{j \in J_i} w_j x_j \leq \bar{c}_{J_i} \\ & x_j \in \{0, 1\}, \quad \text{for } j \in J_i \end{cases}$$

Each $(SK_{J_i}^{\bar{c}_{J_i}})$ problem is associated with a specified class $J_i, i = 1, \ldots, m$, and with a capacity $\bar{c}_{J_i}$. Notice that each $(SK_{J_i}^{\bar{c}_{J_i}})$ represents exactly a Single Knapsack (*SK*) problem (for more details, see Hifi and Sadfi [9]). Moreover, each class J_i is specified by its critical element, which is obtained as follows: (1) we select the elements in the decreasing order of p_j/w_j, and (2) the critical element r_{J_i} is the index of an element of such a class J_i realizing

$$\sum_{k_{J_i}=1}^{r_{J_i}-1} w_{k_{J_i}} \leq \bar{c}_{J_i} \tag{1}$$

$$\sum_{k_{J_i}=1}^{r_{J_i}} w_{k_{J_i}} > \bar{c}_{J_i} \tag{2}$$

$$\sum_{i=1}^{m} \bar{c}_{J_i} \leq c \tag{3}$$

We consider that a tabu list is a vector characterizing the partial critical elements $(r_{J_1}, \ldots, r_{J_m})$. We can show that there exists a bijection between the set of critical elements and the set of all solutions obtained by combining different solutions. In our study, we consider only the critical elements to simplify all the search procedures for KSP.

The description of the solution is such that, for each class $J_i, i = 1, \ldots, m$, all items located before r_{J_i} are set equal to "one" and items located after r_{J_i} are "free." The obtained solution is feasible and is completed by applying the *GH* procedure. At each step of the algorithm, a new solution is generated and the tabu list updated with a new critical vector, and we remove the old critical vector from the tabu list. This tabu list is an array recording the critical elements for a given number of iterations.

4.4. *A neighborhood of the current solution*

In this section, we develop the main principle of the *Single-depth Tabu Search* algorithm (STS). In order to improve each current feasible solution, we run on it a neighborhood search. We recall that each feasible solution contains some critical elements, therefore, by moving some of these elements, we construct a set of solutions representing the neighborhood of the current feasible solution. Each class J_i has a critical element r_{J_i} verifying Eqs. (1)–(3) of Section 4.3. For a given class, figure 2 shows the structure of an eventual partial exact solution, and figure 3 illustrates the configuration of an eventual partial feasible solution.

Let Δ be a nonnegative integer representing a *depth parameter*. The depth parameter allows us to jump backward from r_{J_i} to $r_{J_i} - 1$ and so on for each class J_i until the total depth has been explored. For each class, and with reference to a certain feasible solution,

Element of J_i	$\longrightarrow$	1	2	3	4	5	...	r_{J_i}	...	$n_{J_i}-1$	n_{J_i}
subbit partial solution	$\longrightarrow$	1	1	1	1	1	...	0	...	0	0

Figure 2. Binary representation of a partial KSP solution for the i-th class.

Element of J_i	$\longrightarrow$	1	...	$r_{J_i}-1$	r_{J_i}					...	n_{J_i}
subbit feasible solution	$\longrightarrow$	1	...	1	0	$*$	$*$	$*$	0	...	0

Figure 3. Binary representation of a feasible KSP solution for the i-th class. The symbol $*$ denotes free items not yet fixed in a given class.

we make moves in the current tabu list as follows: (1) let $x_{r_{J_i}-1}$ be the $(r_{J_i}-1)$-*th* item of the *i-th* class, (2) set $x_{r_{J_i}-1}$ equal to zero, and (3) complete the resulting configuration by applying the *GH* procedure.

A set of new solutions is obtained and is called the *neighborhood* of the current solution. A new search is then applied to improve the solution, i.e., the solution with the best value in the neighborhood is selected for which the critical vector is not in the tabu list. This process is iterated until the total depth is completely explored.

The main steps of the STS algorithm are described in figure 4. The algorithm starts by setting the neighborhood to the empty set and by considering that the stopping criterion is represented by a constant *MaxIter*. At each iteration (of the `Main step`) and for each class $i \in \{1, \dots, m\}$, r'_{J_i} is set equal to j, where j denotes the index of the best critical element taken in the following order: $r_{J_i}-1, \dots, r_{J_i}-\Delta$. The current solution is completed according to the new configuration realized with the obtained critical element as $(r_{J_1}-1, \dots, r'_{J_i}, \dots, r_{J_m}-1)$. In this case, a neighborhood of a solution is then obtained for the considered depth Δ. In fact, for each constructed solution $m\Delta$, neighbors are considered. The best solution is updated if one of the neighbors improves the current solution and does not belong to the tabu list. For the current iteration, a new solution is obtained and is considered as tabu relating to its critical element. The tabu list is then composed by the current best critical vector element if we consider all the classes J_i, $i = 1, \dots, m$. The same process is iterated by considering the new obtained solution and so on.

We can see that the STS has a complexity of $O(mn)$. On the one hand, at each step of the `loop for` of the `main step` of the algorithm of figure 4, the function `FindNeighborhood-`

```
INPUT : An instance of the KSP;
OUTPUT : A suboptimal solution value BestSol;
Initialization:
1.  Sol ⟵ procedure GH();         /* Sol is the initial solution produced by GH */
2.  BestSol ⟵ Sol;                /* BestSol is the best current solution obtained up to now */
3.    Call InitTabu(TabuList);    /* InitTabu is applied in order to initialize the tabu list */
4.    Call InitDepth(Δ);          /* InitDepth is used in order to fix the depth parameter */

Main step:
For iteration = 0 to MaxIter do   /* the maximum number of iterations is fixed to MaxIter */
1.  Neighborhood ⟵ FindNeighborhood(Sol, Δ);
      /* FindNeighborhood is applied for constructing a neighborhood of the current solution Sol */
2.  Sol ⟵ FindBestSol(Neighborhood \ TabuList);
      /* FindBestSol selects the best solution, using an aspiration criterion , over the neighborhood */
3.    Call UpdateTabu(Sol, TabuList);
      /* UpdateTabu is called for updating the tabu list and the best current solution */
4.  if (BestSol ≤ Sol) then Sol ⟵ BestSol;
End_For
```

Figure 4. The single-depth tabu search approach.

(*Sol*, Δ) takes $\Delta \times O(mn)$ operations, where Δ is the constant representing the depth parameter; therefore, for a maximum number of iterations *MaxIter*, we have *MaxIter* $\times$ $\Delta \times O(mn)$ operations. On the other hand, the `FindBestSol`(*Neighborhood*$\backslash$*TabuList*) procedure has the same complexity, therefore, the complexity of STS is evaluated to $O(mn)$.

In our method, recency-based memory and frequency-based memory are both concentrated in depth exploration and the search for other solutions, i.e., the process starts by making an inventory of candidate solutions, in addition to exploring the total depth. This inventory is treated in the tabu list. The quality component consists of identifying and discriminating elements that seem to improve on the current solution by building its neighborhood. Finally, the influence component tries to capture the best candidates from the beginning of the search process until the best solution is selected with reference to the constructed neighborhood. Locally, the obtained solution could be the worst, but the tabu list represents the history of the process of the search, so attractive areas could then be explored to create more improvements.

4.5. Intensification and diversification strategies

In this section, we propose an improved version of the STS algorithm. The improved version is called *Multiple-depth Tabu Search*, denoted MTS, in which two different depth parameters are used. Generally, two highly important components of TS are intensification and diversification strategies. Intensification strategies are based on modifying choice rules to encourage move combinations and solution features that have historically been found to be good. These strategies may also initiate a return to attractive regions in order to search them more thoroughly. On the other hand, the diversification stage consists of encouraging the search process to examine unvisited regions and to generate solutions that differ in various significant ways from those seen before.

In our study, we have considered a combination of diversification and intensification strategies. This proposed strategy seems an appropriate way for the KSP to select good solutions. The strategy considers two depths called *right depth* (denoted Δ_r) and *left depth* (denoted Δ_l). Each one explores the neighborhood in the opposite direction from the other. In addition, Δ_r is at least two times Δ_l. This setup allows the search to explore the entire *left neighborhood* and to attend to the entire *right neighborhood*. The strategy is based on making combinations of all solutions never visited before, and it modifies choice rules to encourage move combinations and solution features historically found to be good. The strategy may also initiate a return to attractive regions to search them more thoroughly. Such an approach can be based on generating subassemblies of solution components that are then fleshed out to produce full solutions.

In figure 5, we represent an intensified-diversified strategy applied by the MTS algorithm. The symbol $*$ denotes the explored area, with free items and those not fixed yet, for a

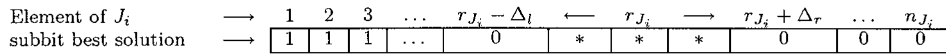

Figure 5. Binary representation of an MTS partial KSP solution for the i-th class.

given first subsolution. Once a Δ_l is fixed, the entire right neighborhood is explored from $r_{J_i} + 1$ to $r_{J_i} + \Delta_r$. The obtained set of all combinations of subsolutions is the constructed neighborhood for the current solution. Furthermore, for each class J_i, if any improved subsolution is obtained, then it is a tabu subsolution. The generalization to the solution is reached by treating all the classes.

4.6. *Using a second neighborhood*

4.6.1. An overview of the MTS approach. The main idea of the MTS approach remains the same as for the STS approach. Here, we consider two different depths, namely, one left-direction depth and another right-direction depth; in other words, we consider two critical areas. At the beginning, we fix the left area depth running heuristic and a *pseudo*-STS algorithm. We then run a right neighborhood search and explore the entire corresponding depth. Then a combination of solutions for the first fixed right depth is generated. A *pseudo*-STS is run for this combination. This principle is applied for the entire left depth area, and we explore, for such a fixed left depth, the total right corresponding depth area. This procedure seems much more interesting in investigating more regions, and it gives many more good solutions, in the sense that it improves the solutions given by the single-depth strategy (see Section 5). Note here that we can also choose more than two depths, but the computational time will increase significantly. Furthermore, with two depths, limited computational experience showed that the MTS approach produced good results within a reasonable computational time.

4.6.2. The main principle of the search process. The main steps of the MTS algorithm are given in figure 4 by replacing the parameter Δ by the pair (Δ_l, Δ_r) in the procedure `InitDepth`(•) (line 4 of `Initialization`) and in the procedure `FindNeighborhood`(*Sol*, •) (line 2 of the `Main step`). The MTS algorithm has the same idea in building neighborhoods and searching for improved solutions. The difference consists in the manipulation of two types of depths for exploring new areas: (1) a right area and (2) a left area. At the beginning, the neighborhood set is initialized to the empty set. At each iteration and for each class, we assign to r'_{J_i} the index j that realizes the best critical element. In this case, the current configuration (considering the new critical element) is denoted by $(r_{J_1} - 1, \ldots, r'_{J_i}, \ldots, r_{J_m} - 1)$. Each considered depth Δ represents a solution—one that can be characterized by its neighborhood. By considering all depths, we obtain a total of $m \times \Delta_r \times \Delta_l$ neighbors for each solution. Note that this outcome is much richer than the one obtained by applying the STS algorithm. The best solution is updated if one of the neighbors improves the current solution and if its critical elements do not belong to the tabu list. At each iteration, a new solution is obtained, and it is considered to be tabu. The tabu list is composed of the current best critical vector for the considered classes. The newly obtained solution is used in the next iterations to try for another local improvement.

We can show that this algorithm has a complexity evaluated to $O(nm)$. Indeed, we just note that the `FindNeighborhood`(*Sol*, Δ_r, Δ_l) (resp. `FindBestSol`(*Neighborhood* \ *TabuList*)) procedure takes $\Delta_r \times \Delta_l \times O(nm)$ operations and that, for a constant $\Delta_r \times \Delta_l \times D$, the MTS algorithm has a complexity of $O(nm)$.

5. Computational results

The proposed algorithm was coded in `C` and executed on an `UltraSparc10` (250 MHz and with 128 Mo of RAM). Our computational study was conducted on 240 problem instances of various sizes and densities. These test problem instances (detailed in Table 1) are standard and publicly available,[1] and their optimal solution values are known and taken from [9]. These instances are divided into two different sets. The first set represents 168 "uncorrelated" instances, and the second one contains 72 "strongly correlated" instances.

The "uncorrelated" instances are generated as follows: for each instance, the number m (classes) is taken in the interval [2, 50], the number n (variables) is taken from [1000, 20000], and w_j and p_j are mutually independent and are uniformly taken from [1, 100] and [1, 50], respectively. The capacity of the KSP c is equal to $\lfloor (\sum_{j=1}^{n} w_j)/2 \rfloor$, and the cardinality of each class $J_i, i = 1, \ldots, m$, is in $[1, n - m + 1]$. The other 72 instances (the "correlated" instances) are generated in the same way as the "uncorrelated" ones, but the profit p_j associated with the objects have been taken to be equal to $w_j + 100$ for $j = 1, \ldots, n$.

Table 1. Test problem details: $1 \leq x \leq 4$.

Inst.	n	m	Inst.	n	m	Inst.	n	m
A02.x	1000	2	B02.x	2500	2	C02.x	5000	2
D02.x	7500	2	E02.x	10000	2	F02.x	20000	2
A05.x	1000	5	B05.x	2500	2	C05.x	5000	2
D05.x	7500	5	E05.x	10000	2	F05.x	20000	2
A10.x	1000	10	B10.x	2500	10	C10.x	5000	10
D10.x	7500	10	E10.x	10000	2	F10.x	20000	10
A20.x	1000	20	B20.x	2500	20	C20.x	5000	20
D20.x	7500	20	E20.x	10000	20	F20.x	20000	20
A30.x	1000	30	B30.x	2500	30	C30.x	5000	30
D30.x	7500	30	E30.x	10000	30	F30.x	20000	30
A40.x	1000	40	B40.x	2500	40	C40.x	5000	40
D40.x	7500	40	E40.x	10000	40	F40.x	20000	40
A50.x	1000	50	B50.x	2500	50	C50.x	5000	50
D50.x	7500	50	E50.x	10000	50	F50.x	20000	50
A02C.x	1000	2	B02C.x	2500	2	C02C.x	5000	2
D02C.x	7500	2	E02C.x	10000	2	F02C.x	20000	2
A05C.x	1000	5	B05C.x	2500	5	C05C.x	5000	5
D05C.x	7500	5	E05C.x	10000	5	F05C.x	20000	5
A10C.x	1000	10	B10C.x	2500	10	C10C.x	5000	10
D10C.x	7500	10	E10C.x	10000	10	F10C.x	20000	10

Table 2. Representation of the quality results realized by the three versions of the algorithm.

	Single depth without tabu list			Single depth with tabu list			Multiple depth with tabu list		
Instances (no. of *nb*)	Optima	%Av. Rat.	Worst	Optima	%Av. Rat.	Worst	Optima	%Av. Rat.	Worst
Uncorrelated (168)	18	0.210	1.455	81	0.051	0.476	129	0.019	0.316
Correlated (72)	5	0.187	0.722	25	0.014	0.097	50	0.003	0.029

5.1. The summary results

In a preliminary experiment, we have solved the problem instances by considering several versions of the algorithm. A summary of the obtained results appears in Table 2, where we report, for each version, the number of instances solved at the optimum (denoted by *Optima*) and the Average Percentage Deviation (*Av. P.D.*), representing the average gap $Av.\ P.D. = \sum_{i=1}^{\ell}(\frac{Opt_i - A_i}{Opt_i} \times 100\%)/\ell$ (where ℓ denotes the number of instances, A_i represents the solution value obtained by applying the algorithm on the instance i, and Opt_i is the optimal solution value of this instance). We have also reported the worst-case percentage deviation of each group, denoted *Worst*.

First, we have considered a simple version that combines the GH procedure and a simple Local Search, called herein GH-LS. In this case, we have removed the tabu list. The second version of the algorithm, called STS herein, represents the algorithm GH-LS, in which a tabu list is introduced. Finally, we have considered a third version representing the MTS algorithm. The implementation of the different versions of the algorithm involves some decisions: the way to set the number of iterations, the right values of the left-depth Δ_l (or Δ) and right-depth Δ_r, and the length of the tabu list. Several strategies have been explored, and we have chosen the strategy that produced good results without large computational time. In our study, we have retained the following strategies. The stopping criterion for the GH-LS and STS algorithms was set to a maximum number of iterations of 500 for the uncorrelated (resp. 300 for the correlated) problem instances. The depth parameter Δ was set at five. For the MTS algorithm, the maximum number of iterations was set, respectively, at 300 for the uncorrelated and at 200 for the correlated ones. The left-depth Δ_l was set at five and the right-depth Δ_r was doubled. Furthermore, for the STS and MTS versions, the length of the tabu list varies dynamically. Indeed, if m^* is the number of the different classes, then the length is automatically and randomly taken in the integer interval $[\lfloor\sqrt{m^*}\rfloor + 1, \lfloor\sqrt{m^*}\rfloor + 5]$. The change of the tabu list is made after 25 iterations if the best current solution has not improved. These limits allows us to keep the average computing times below two minutes of CPU.

The quality of the results obtained by the three versions of the algorithm are given in Table 2. Examining Table 2, we observe the following:

- The GH-LS algorithm produces reasonable quality results (see columns 2, 3, and 4 of Table 2)—on average, 0.21% (for the uncorrelated instances) and 0.187% (for the correlated instances) from the optimum—but it can give poor results in some instances, with a worst-case result of 1.455%. This finding means that GH-LS gives only a feasible

solution without knowing how to improve it, since the tabu list was canceled. The history of tabu possible candidates is not taken into account, so the search process cannot escape from some local optima.

- The results of the GH-LS algorithm are improved by adding the tabu list (see columns 5, 6, and 7 of Table 2). The quality of the obtained solutions is higher, with an overall percentage of 0.051% (resp. 0.014%) for the uncorrelated (resp. correlated) instances. These results are better than those obtained by directly using the GH-LS algorithm. More concretely, if we use the STS algorithm, the number of optimal solutions obtained increases from 18 to 81 (resp. from 5 to 25) for the uncorrelated (resp. correlated) instances, and globally, the percentage of the attainable optimal solutions increases from 9.58% to 44.17%. In this case, we can show that using the tabu strategy is efficient and improves several solutions.
- The MTS algorithm, with multiple depth parameters, sometimes needs longer computing times (as shown in Table 2, columns 8, 9, and 10), but it obtains very high quality results even in the worst cases: 179 out of 240 optimal solutions are attained, and the overall percentage with respect to the optimum is 0.011%. These good results increase for both type of instances, (uncorrelated with a percentage deviation of 0.019% and 129 optima out of 168 instances) and correlated (with a percentage deviation of 0.003% and 50 optima out of 72 instances).

5.2. The test details

In this section, we present a detailed experimental study of the MTS version of the algorithm. As shown in Table 2, the MTS algorithm outperforms the others in the sense that it gives better average percentage deviation. For each instance, we report (see Tables 3–5) the Percentage Deviation (P.D.) and the execution time (CPU), which is the total time (measured in seconds) that MTS algorithm takes before termination. We use also the symbol $\triangleright$ if the algorithm produces the optimal solution and the symbol $\diamond$ if the MTS algorithm improves the solution value produced by the STS algorithm. The results are shown in Tables 3 and 4 for uncorrelated instances and in Table 5 for the correlated ones.

Examining Tables 3–5, we observe the following:

1. For the first set of problems representing the uncorrelated instances (see Tables 3 and 4), we note that the MTS algorithm performs better than the STS one (see columns under "Best" marked with the symbol $\diamond$). Indeed, the MTS version improves 75 solution values out of 168, which represents a percentage of 44.64%. Figure 6 shows the dispersion P.D.s of the improved solutions. Note that the average best-solution time (columns under CPU of Tables 3 and 4), realized by the MTS algorithm, is under 100 seconds for all uncorrelated instances.
2. For the second set of problems representing the correlated instances (see Table 5), we note that the phenomenon is the same as for the uncorrelated ones. In this case, the algorithm performs better than the other versions by improving 43 solution values out of 72, which represents a percentage of 59.72% (figure 7 shows these obtained improvements). We can also remark that the average best-solution time (columns CPU of Table 5), realized by both algorithms, is under 100 seconds for all correlated instances.

Table 3. Performance of the multiple depth tabu search (MTS) approach on the uncorrelated instances with $m \leq 10$ (classes).

Inst.	Opt.	Best	P.D.	CPU	Inst.	Opt.	Best	P.D.	CPU
A02.1	20490	20490$^{\diamond}$	$\triangleright$	1.9	B02.1	50803	50803	$\triangleright$	6.2
A02.2	20419	20419	$\triangleright$	1.9	B02.2	50136	50136	$\triangleright$	5.5
A02.3	20889	20888	0.005	2.0	B02.3	50873	50873$^{\diamond}$	$\triangleright$	5.7
A02.4	20564	20564$^{\diamond}$	$\triangleright$	1.9	B02.4	52143	52143	$\triangleright$	6.0
A05.1	8071	8058$^{\diamond}$	0.161	3.3	B05.1	20371	20370$^{\diamond}$	0.005	5.3
A05.2	7995	7990$^{\diamond}$	0.063	2.8	B05.2	20349	20349$^{\diamond}$	$\triangleright$	5.6
A05.3	7960	7960$^{\diamond}$	$\triangleright$	3.3	B05.3	20424	20424$^{\diamond}$	$\triangleright$	5.8
A05.4	8115	8115$^{\diamond}$	$\triangleright$	2.6	B05.4	20376	20376$^{\diamond}$	$\triangleright$	5.9
A10.1	4054	4054	$\triangleright$	3.5	B10.1	10047	10043$^{\diamond}$	0.040	10.8
A10.2	3525	3525	$\triangleright$	2.7	B10.2	9905	9905$^{\diamond}$	$\triangleright$	8.1
A10.3	4087	4087$^{\diamond}$	$\triangleright$	3.1	B10.3	10129	10129$^{\diamond}$	$\triangleright$	7.4
A10.4	4037	4037$^{\diamond}$	$\triangleright$	5.3	B10.4	10360	10360$^{\diamond}$	$\triangleright$	9.1
C02.1	102284	102284	$\triangleright$	13.4	D02.1	153401	153401	$\triangleright$	23.5
C02.2	101565	101565	$\triangleright$	13.2	D02.2	152374	152374	$\triangleright$	23.8
C02.3	101551	101551	$\triangleright$	13.3	D02.3	153455	153455	$\triangleright$	24.5
C02.4	104568	104568$^{\diamond}$	$\triangleright$	15.2	D02.4	155556	155556	$\triangleright$	24.1
C05.1	40564	40564$^{\diamond}$	$\triangleright$	11.9	D05.1	61486	61480	0.010	19.0
C05.2	40798	40795$^{\diamond}$	0.007	12.8	D05.2	61000	61000$^{\diamond}$	$\triangleright$	19.1
C05.3	40438	40438$^{\diamond}$	$\triangleright$	10.7	D05.3	60690	60690$^{\diamond}$	$\triangleright$	19.4
C05.4	40449	40449$^{\diamond}$	$\triangleright$	11.4	D05.4	60750	60750	$\triangleright$	19.3
C10.1	20391	20391$^{\diamond}$	$\triangleright$	12.7	D10.1	30574	30569$^{\diamond}$	0.016	18.5
C10.2	20252	20252$^{\diamond}$	$\triangleright$	14.4	D10.2	30460	30460	$\triangleright$	24.5
C10.3	20213	20213$^{\diamond}$	$\triangleright$	15.8	D10.3	30619	30619$^{\diamond}$	$\triangleright$	19.6
C10.4	20858	20854$^{\diamond}$	0.019	14.5	D10.4	31029	31029$^{\diamond}$	$\triangleright$	19.8
E02.1	203577	203577	$\triangleright$	36.4	F02.1	409305	409305	$\triangleright$	98.8
E02.2	204750	204750	$\triangleright$	38.2	F02.2	409818	409818	$\triangleright$	99.7
E02.3	204462	204462	$\triangleright$	37.5	F02.3	409741	409741$^{\diamond}$	$\triangleright$	96.4
E02.4	207203	207203	$\triangleright$	36.7	F02.4	406213	406213$^{\diamond}$	$\triangleright$	91.6
E05.1	81275	81275$^{\diamond}$	$\triangleright$	27.6	F05.1	163514	163510$^{\diamond}$	0.002	65.5
E05.2	81240	81238$^{\diamond}$	0.002	29.4	F05.2	162566	162566	$\triangleright$	64.7
E05.3	81956	81956$^{\diamond}$	$\triangleright$	27.3	F05.3	163850	163850$^{\diamond}$	$\triangleright$	72.5
E05.4	81196	81194$^{\diamond}$	0.002	29.9	F05.4	163202	163202	$\triangleright$	53.6
E10.1	40681	40680$^{\diamond}$	0.002	24.5	F10.1	81739	81739	$\triangleright$	50.7
E10.2	40828	40828	$\triangleright$	26.7	F10.2	81957	81952$^{\diamond}$	0.006	52.4
E10.3	40839	40839$^{\diamond}$	$\triangleright$	24.0	F10.3	81917	81917$^{\diamond}$	$\triangleright$	54.0
E10.4	41406	41406	$\triangleright$	25.0	F10.4	81168	81168	$\triangleright$	41.7

Table 4. Performance of the multiple depth tabu search algorithm on the uncorrelated instances with $m \geq 20$ (classes).

Inst.	Opt.	Best	P.D.	CPU	Inst.	Opt.	Best	P.D.	CPU
A20.1	1989	1987$^{\diamond}$	0.100	8.6	A30.1	1088	1088	$\triangleright$	16.4
A20.2	1465	1465	$\triangleright$	9.8	A30.2	747	747	$\triangleright$	20.8
A20.3	2001	2001$^{\diamond}$	$\triangleright$	4.4	A30.3	1277	1277	$\triangleright$	10.8
A20.4	1972	1972$^{\diamond}$	$\triangleright$	8.1	A30.4	1198	1198	$\triangleright$	10.7
A40.1	712	712	$\triangleright$	10.7	A50.1	550	550	$\triangleright$	55.9
A40.2	595	595	$\triangleright$	37.8	A50.2	459	459	$\triangleright$	43.8
A40.3	716	716	$\triangleright$	31.6	A50.3	536	536	$\triangleright$	52.9
A40.4	668	668	$\triangleright$	36.0	A50.4	489	489	$\triangleright$	59.8
B20.1	4997	4991$^{\diamond}$	0.120	12.3	B30.1	2525	2525	$\triangleright$	22.3
B20.2	4164	4164	$\triangleright$	10.3	B30.2	3123	3123	$\triangleright$	12.2
B20.3	4996	4981	0.300	2.7	B30.3	3218	3218	$\triangleright$	22.5
B20.4	5115	5115$^{\diamond}$	$\triangleright$	13.0	B30.4	2716	2716	$\triangleright$	12.1
B40.1	2173	2173	$\triangleright$	30.7	B50.1	1811	1811	$\triangleright$	58.8
B40.2	2177	2177	$\triangleright$	40.3	B50.2	1615	1615	$\triangleright$	40.7
B40.3	2181	2181	$\triangleright$	20.4	B50.3	1511	1511	$\triangleright$	62.5
B40.4	2057	2057	$\triangleright$	39.5	B50.4	1780	1780	$\triangleright$	40.9
C20.1	10113	10113	$\triangleright$	17.3	C30.1	6719	6698	0.313	26.6
C20.2	10066	10047	0.189	25.1	C30.2	6698	6690$^{\diamond}$	0.119	31.2
C20.3	10079	10065	0.139	17.4	C30.3	6594	6594$^{\diamond}$	$\triangleright$	48.0
C20.4	10377	10377$^{\diamond}$	$\triangleright$	20.5	C30.4	6206	6206	$\triangleright$	12.5
C40.1	4644	4644	$\triangleright$	24.5	C50.1	3935	3935	$\triangleright$	28.6
C40.2	4846	4846	$\triangleright$	41.6	C50.2	3992	3980$^{\diamond}$	0.301	0.6
C40.3	4588	4588	$\triangleright$	33.3	C50.3	3633	3633	$\triangleright$	54.0
C40.4	5122	5122$^{\diamond}$	$\triangleright$	34.9	C50.4	3994	3994	$\triangleright$	48.6
D20.1	15276	15272$^{\diamond}$	0.026	23.4	D30.1	10129	10115$^{\diamond}$	0.138	33.6
D20.2	15151	15151	$\triangleright$	27.8	D30.2	10103	10103$^{\diamond}$	$\triangleright$	45.1
D20.3	15256	15253$^{\diamond}$	0.020	25.7	D30.3	10153	10146$^{\diamond}$	0.069	29.8
D20.4	15468	15447	0.136	27.6	D30.4	9591	9591	$\triangleright$	12.8
D40.1	7606	7582	0.316	38.3	D50.1	6057	6057	$\triangleright$	1.0
D40.2	7505	7505$^{\diamond}$	$\triangleright$	43.0	D50.2	5797	5797	$\triangleright$	62.0
D40.3	7074	7074	$\triangleright$	22.9	D50.3	5407	5407	$\triangleright$	54.6
D40.4	7663	7652$^{\diamond}$	0.144	45.6	D50.4	6131	6131$^{\diamond}$	$\triangleright$	0.9
E20.1	20274	20274	$\triangleright$	31.6	E30.1	13438	13422	0.119	56.2
E20.2	20382	20382	$\triangleright$	33.7	E30.2	13556	13556$^{\diamond}$	$\triangleright$	34.2
E20.3	20368	20358$^{\diamond}$	0.049	28.0	E30.3	13590	13590$^{\diamond}$	$\triangleright$	53.0
E20.4	20634	20634$^{\diamond}$	$\triangleright$	38.8	E30.4	13200	13200	$\triangleright$	13.5

(*Continued on next page*).

Table 4. (*Continued*).

Inst.	Opt.	Best	P.D.	CPU	Inst.	Opt.	Best	P.D.	CPU
E40.1	10098	10098	▷	49.4	E50.1	8081	8079◇	0.025	1.4
E40.2	9838	9838	▷	34.3	E50.2	8081	8079	0.025	1.4
E40.3	10150	10150◇	▷	64.2	E50.3	8111	8111	▷	1.4
E40.4	10260	10248◇	0.117	48.4	E50.4	8195	8195	▷	1.5
F20.1	40884	40884	▷	69.6	F30.1	27217	27217	▷	63.7
F20.2	40926	40926◇	▷	67.3	F30.2	27250	27244	0.022	61.0
F20.3	40936	40936◇	▷	58.6	F30.3	27223	27223◇	▷	62.2
F20.4	40528	40528	▷	52.5	F30.4	26938	26938	▷	79.8
F40.1	20393	20390◇	0.015	83.9	F50.1	16262	16259◇	0.018	4.4
F40.2	20425	20425	▷	97.0	F50.2	16291	16288◇	0.018	3.5
F40.3	20428	20428◇	▷	70.0	F50.3	16326	16326	▷	3.5
F40.4	20218	20218	▷	73.0	F50.4	16123	16114◇	0.056	79.7

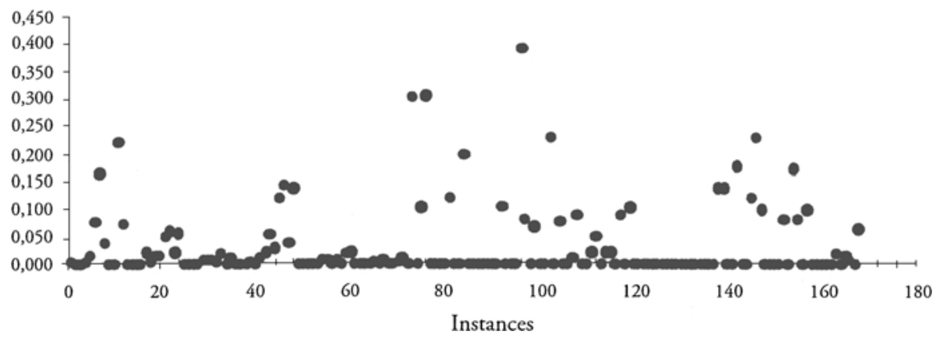

Figure 6. Representation of the percentage improvement (of the P.D.s) when the MTS algorithm is applied: the uncorrelated instances.

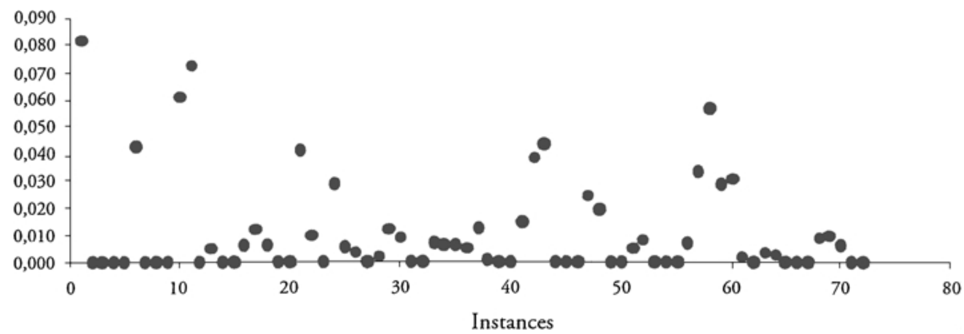

Figure 7. MTS algorithm: representation of the improvement of the P.D.s for the correlated instances.

Table 5. Performance of the multiple depth tabu search approach on the "correlated" instances.

No. of instances	Opt.	Sol.	P.D.	CPU	No. of instances	Opt.	Sol.	P.D.	CPU
A02C.1	41492	41492$^{\diamond}$	$\triangleright$	3.0	D02C.1	312223	312217	0.002	14.6
A02C.2	41397	41397	$\triangleright$	1.6	D02C.2	311947	311947	$\triangleright$	14.6
A02C.3	41565	41565	$\triangleright$	2.0	D02C.3	311690	311690$^{\diamond}$	$\triangleright$	14.7
A02C.4	41556	41556	$\triangleright$	1.6	D02C.4	311419	311417$^{\diamond}$	0.001	14.5
A05C.1	16554	16554	$\triangleright$	1.9	D05C.1	124794	124782	0.010	12.1
A05C.2	16561	16561$^{\diamond}$	$\triangleright$	1.7	D05C.2	124758	124758	$\triangleright$	11.9
A05C.3	16590	16587	0.018	3.1	D05C.3	124669	124669	$\triangleright$	11.9
A05C.4	16584	16584	$\triangleright$	3.9	D05C.4	124529	124529$^{\diamond}$	$\triangleright$	11.6
A10C.1	8245	8245	$\triangleright$	2.9	D10C.1	62334	62334$^{\diamond}$	$\triangleright$	9.6
A10C.2	8203	8203$^{\diamond}$	$\triangleright$	4.3	D10C.2	62311	62311$^{\diamond}$	$\triangleright$	10.0
A10C.3	8250	8248$^{\diamond}$	0.024	4.6	D10C.3	62289	62280$^{\diamond}$	0.014	9.8
A10C.4	8255	8255	$\triangleright$	5.6	D10C.4	62216	62211$^{\diamond}$	0.008	10.3
B02C.1	103672	103672$^{\diamond}$	$\triangleright$	2.8	E02C.1	416396	416396$^{\diamond}$	$\triangleright$	23.4
B02C.2	103572	103555$^{\diamond}$	0.016	3.2	E02C.2	415426	415426$^{\diamond}$	$\triangleright$	24.2
B02C.3	104034	104034	$\triangleright$	2.9	E02C.3	415470	415470	$\triangleright$	24.1
B02C.4	104031	104031	$\triangleright$	2.7	E02C.4	415341	415334$^{\diamond}$	0.002	23.6
B05C.1	41393	41393$^{\diamond}$	$\triangleright$	2.7	E05C.1	166484	166474$^{\diamond}$	0.006	14.1
B05C.2	41437	41425$^{\diamond}$	0.029	3.1	E05C.2	166126	166126$^{\diamond}$	$\triangleright$	14.2
B05C.3	41580	41580$^{\diamond}$	$\triangleright$	2.7	E05C.3	166116	166094	0.013	14.5
B05C.4	41561	41561	$\triangleright$	3.7	E05C.4	166062	166062	$\triangleright$	13.7
B10C.1	20657	20657	$\triangleright$	6.3	E10C.1	83216	83199$^{\diamond}$	0.020	12.4
B10C.2	20648	20648	$\triangleright$	7.2	E10C.2	82995	82985$^{\diamond}$	0.012	15.5
B10C.3	20740	20740$^{\diamond}$	$\triangleright$	6.8	E10C.3	83029	83014$^{\diamond}$	0.018	14.7
B10C.4	20721	20721$^{\diamond}$	$\triangleright$	7.7	E10C.4	82981	82981$^{\diamond}$	$\triangleright$	15.7
C02C.1	207675	207673$^{\diamond}$	0.001	7.6	F02C.1	830622	830611$^{\diamond}$	0.001	82.9
C02C.2	207916	207916	$\triangleright$	8.1	F02C.2	831513	831513	$\triangleright$	82.9
C02C.3	208043	208043	$\triangleright$	10.1	F02C.3	831145	831145$^{\diamond}$	$\triangleright$	83.9
C02C.4	207973	207973$^{\diamond}$	$\triangleright$	10.8	F02C.4	831445	831445$^{\diamond}$	$\triangleright$	85.4
C05C.1	83013	83013$^{\diamond}$	$\triangleright$	6.2	F05C.1	332143	332126	0.005	41.6
C05C.2	83129	83120$^{\diamond}$	0.011	6.5	F05C.2	332527	332527	$\triangleright$	41.9
C05C.3	83185	83185	$\triangleright$	6.6	F05C.3	332402	332388	0.004	42.5
C05C.4	83188	83188	$\triangleright$	5.9	F05C.4	332498	332498$^{\diamond}$	$\triangleright$	40.8
C10C.1	41466	41466$^{\diamond}$	$\triangleright$	7.7	F10C.1	166028	166028$^{\diamond}$	$\triangleright$	29.9
C10C.2	41521	41518$^{\diamond}$	0.007	10.7	F10C.2	166248	166248$^{\diamond}$	$\triangleright$	28.6
C10C.3	41554	41554	$\triangleright$	12.0	F10C.3	166154	166148	0.004	29.8
C10C.4	41554	41554$^{\diamond}$	$\triangleright$	6.5	F10C.4	166192	166192	$\triangleright$	30.4

Table 6. Computational times of the single/multiple depth approach compared to the exact algorithm.

	Single depth approach			Multiple depth approach			Exact approach		
Instances (no. of *nb*)	Min.	Av.	Max.	Min.	Av.	Max.	Min.	Av.	Max.
Uncorrelated (168)	0.3	12.8	83.1	0.6	30.0	109.7	0.4	106.3	1687.5
Correlated (72)	0.2	15.8	74.8	1.6	16.2	85.4	0.8	212.6	1682.4
Average		14.3			23.1			159.5	

We have also compared the computing times consumed by both the STS and MTS approaches to the computing times of the exact algorithm developed in [9]. A summary of the results appears in Table 6. Over all instances, we have reported the minimum (denoted Min.), maximum (denoted Max.), and average computing time (denoted Av.) realized by each algorithm. We note that the STS (resp. MTS) algorithm uses on average 8.96% (resp. 14.48%) of the time consumed by the exact algorithm. We think these later percentages are reasonable for both approaches, since they produce good average percentage deviation. For example, the MTS approach is able to produce high-quality results (the overall P.D. with respect to the optimum is 0.011%), and it attains 74.58% of optimal solutions.

6. Conclusion

In this article, we have proposed an efficient approximate algorithm for solving the knapsack sharing problem. The approach is mainly based upon tabu search and upon combining a single or double depth parameter with a tabu search. The principle of the method is to detect some critical elements of the current feasible solution and to tailor a neighborhood search on that solution. We have used a depth-parameter strategy and designed a heuristic feasibility in order to improve the performance of the algorithm. Computational results indicate that the two versions of the algorithm are able to generate high-quality solutions for the knapsack sharing problem, within reasonable computing times.

Acknowledgment

Many thanks to anonymous referees for their helpful comments and suggestions, which helped to improve the contents and the presentation of this article.

Note

1. From `ftp://panoramix.univ-paris1.fr/pub/CERMSEM/hifi/OR-Benchmark.html`.

References

1. J.R. Brown, "The knapsack sharing," Operations Research, vol. 27, pp. 341–355, 1979.
2. J.R. Brown, "Solving knapsack sharing with general tradeoff functions," Mathematical Programming, vol. 51, pp. 55–73, 1991.

3. P. Chu and J.E. Beasley, "A genetic algorithm for the multidimensional knapsack problem," Journal of Heuristics, vol. 4, pp. 63–86, 1998.
4. A. Freville and G. Plateau, "The 0-1 bidimensional knapsack problem: Toward an efficient high-level primitive tool," Journal of Heuristics, vol. 2, pp. 147–167, 1997.
5. P.C. Gilmore and R.E. Gomory, "The theory and computation of knapsack functions," Operations Research, vol. 13, pp. 879–919, 1966.
6. F. Glover, "Future paths for integer programming and links to artificial intelligence," Computers and Operations Research, vol. 13, pp. 533–549, 1986.
7. F. Glover and M. Laguna, Tabu Search, Kluwer Academic Publishers: Boston, MA, 1997.
8. P. Hansen, "The steepest ascent mildest descent heuristic for combinatorial programming," Presented at the Congress on Numerical Methods in Combinatorial Optimization, Capri, Italy, 1986.
9. M. Hifi and S. Sadfi, "The knapsack sharing problem: An exact algorithm," Journal of Combinatorial Optimization, vol. 6, pp. 35–45, 2002.
10. T. Kuno, H. Konno, and E. Zemel, "A linear-time algorithm for solving continuous maximum knapsack problems," Operations Research Letters, vol. 10, pp. 23–26, 1991.
11. H. Luss, "Minmax resource allocation problems: Optimization and parametric analysis," European Journal of Operational Research, vol. 60, pp. 76–86, 1992.
12. S. Martello and P. Toth, Knapsack Problems: Algorithms and Computer Implementation, John Wiley: New York, 1990.
13. S. Martello and P. Toth, "Upper bounds and algorithms for hard 0-1 knapsack problems," Operations Research, vol. 45, pp. 768–778, 1997.
14. J.S. Pang and C.S. Yu, "A min-max resource allocation problem with substitutions," European Journal of Operational Research, vol. 41, pp. 218–223, 1989.
15. D. Pisinger, "A minimal algorithm for the 0-1 knapsack problem," Operations Research, vol. 45, pp. 758–767, 1997.
16. M. Syslo, N. Deo, and J. Kowalik, Discrete Optimization Algorithms, Prentice-Hall, 1983.
17. C.S. Tang, "A max-min allocation problem: Its solutions and applications," Operations Research, vol. 36, pp. 359–367, 1988.
18. T. Yamada and M. Futakawa, "Heuristic and reduction algorithms for the knapsack sharing problem," Computers and Operations Research, vol. 24, pp. 961–967, 1997.
19. T. Yamada, M. Futakawa, and S. Kataoka, "Some exact algorithms for the knapsack sharing problem," European Journal of Operational Research, vol. 106, pp. 177–183, 1998.