A Reactive Local Search-Based Algorithm for the Multiple-Choice Multi-Dimensional Knapsack Problem

Abdelkader Sbihi, Michrafy Mustapha, Mhand Hifi

▶ To cite this version:

Abdelkader Sbihi, Michrafy Mustapha, Mhand Hifi. A Reactive Local Search-Based Algorithm for the Multiple-Choice Multi-Dimensional Knapsack Problem. Computational Optimization and Applications, 2006, 33 (2-3), pp.271-285. 10.1007/s10589-005-3057-0 . hal-00125573

HAL Id: hal-00125573
https://hal.science/hal-00125573

Submitted on 23 Jan 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Computational Optimization and Applications
© 2005 Springer Science + Business Media, Inc. Manufactured in The Netherlands.
DOI: 10.1007/s10589-005-3057-0

A Reactive Local Search-Based Algorithm for the Multiple-Choice Multi-Dimensional Knapsack Problem

M. HIFI
M. MICHRAFY
A. SBIHI
LaRIA, Laboratoire de Recherche en Informatique d'Amiens, 5 rue du Moulin Neuf, 80000 Amiens, France

Received April 16, 2003; Revised October 12, 2004; Accepted January 12, 2005

Published online: 18 October 2005

Abstract. In this paper, we approximately solve the multiple-choice multi-dimensional knapsack problem. We propose an algorithm which is based upon reactive local search and where an explicit check for the repetition of configurations is added to the local search. The algorithm starts by an initial solution and improved by using a fast iterative procedure. Later, both *deblocking* and *degrading* procedures are introduced in order (i) to escape to local optima and, (ii) to introduce diversification in the search space. Finally, a memory list is applied in order to forbid the repetition of configurations. The performance of the proposed approaches has been evaluated on several problem instances. Encouraging results have been obtained.

Keywords: combinatorial optimization, heuristics, knapsacks, reactive local search

1. Introduction

In this paper, we deal with a particular 0–1 Knapsack Problem (KP) known as Multiple-Choice Multi-Dimensional Knapsack Problem (MMKP). MMKP concerns many practical problems in the real life as the service level agreement, the model of allocation resources and, the dynamic adaptation of system of resources for multimedia multi-sessions (for more details, one can refer to Khan et al. [8] and Khan [7]).

In the MMKP, we have a multi-constrained knapsack of a capacity vector or available resources, namely $C = (C^1, C^2, \ldots, C^m)$, and a set $J = (J_1, \ldots, J_i, \ldots, J_n)$ of items divided into n disjoint classes, where each class J_i, $i = 1, \ldots, n$, has $r_i = |J_i|$ items. Each item $j, j = 1, \ldots, r_i$, of class J_i has a nonnegative profit value v_{ij}, and requires resources given by the weight vector $W_{ij} = (w_{ij}^1, w_{ij}^2, \ldots, w_{ij}^m)$. Each weight component w_{ij}^k (with $1 \leq k \leq m$, $1 \leq i \leq n$, $1 \leq j \leq r_i$) also has a nonnegative value. The problem is to fill the knapsacks with exactly one item from each class in order to maximize the total profit value of the choice, such that the capacity constraints are satisfied. By the total profit value of the choice, we mean the sum of the profits of items fixed in the multi-constrained knapsack. The MMKP

can be formulated as follows:

$$(MMKP)\quad \begin{cases} \text{maximize} & Z = \sum_{i=1}^{n}\sum_{j=1}^{r_i} v_{ij}x_{ij} & & \\ \text{subject to} & \sum_{i=1}^{n}\sum_{j=1}^{r_i} w_{ij}^k x_{ij} \le C^k, & k \in \{1,\ldots,m\} & \\ & \sum_{j=1}^{r_i} x_{ij} = 1, & i \in \{1,\ldots,n\} & \\ & x_{ij} \in \{0,1\}, & i \in \{1,\ldots,n\}, & j \in \{1,\ldots,r_i\} \end{cases}$$

where x_{ij} is either 0, implying item j of the i-th class J_i is not picked, or 1 implying item j of the i-th class is picked.

In this paper, we propose an algorithm for MMKP, where an explicit check for the repetition of configurations is combined with a fast local search. In the proposed approach, the appropriate size of the list is learned by reacting to the occurrence of cycles. In addition, if the search repeats an excessive number of solutions excessively often, then the search is diversified by making a number of degrading operations with respect to the cycle length. The reactive mechanism is compared to a "simple" Tabu Search (TS), as a storage list is introduced, that forbids the repetition of configurations.

The proposed approach can be summarized as follows: (i) starting with an initial solution for the MMKP, obtained by applying a fast constructive procedure, (ii) improving the current solution by running a complementary constructive procedure which applies a swapping criterion and, (iii) using the reactive strategy composed by *deblocking* and *degrading* procedures. Finally, a tabu list is introduced in a modified version of the algorithm. The last list is used in order to avoid some cycling during the search process.

The remainder of the paper is organized as follows. In Section 2, we present a brief reference of some sequential exact and approximate algorithms for knapsack problem variants. The concept of the local search and the proposed algorithm are presented in Section 4. In Sections 4.1 and 4.2, we describe the solution representation and how we obtain the starting solution. The main steps of the two versions of the algorithm are detailed in Sections 4.3 and 4.4. Finally, in Section 5, the performance of both versions is tested on a set of problem instances extracted from the literature and other large randomly generated instances.

2. Related works

There exist several approaches for solving KP and its variants. For the (un)bounded single constraint KP, a large variety of solution methods have been proposed (see Martello et al. [10], Balas and Zemel [1], Fayard and Plateau [4] and Pisinger [1]). The problem has been solved by dynamic programming, tree search procedures and hybrid approaches (for more details on knapsack variants, one can refer to the monograph by Kellerer et al. [6]).

The Multi-Dimensional Knapsack Problem (MDKP) (see Chu and Beasley [2]) is one kind of KP where the constraints are multidimensional. The Multiple-Choice Knapsack (MCKP) (see Pisinger [13]) is another variant of KP where the picking criterion of items is more restrictive. For MDKP, Toyoda [16] used the aggregate resource consumption. The solution of the MDKP needs iterative picking of items until the resource constraint is violated. Other approaches have been used with great success, achieved via the application of local search techniques and metaheuristics to MDKP. Among these approaches, we can cite the tabu search, genetic algorithms, simulated annealing and hybrid algorithms.

To our knowledge, very few papers dealing directly with the MMKP are available. Moser et al. [11] have designed an approach based upon the concept of graceful degradation from the most valuable items based on Lagrange multipliers. Khan et al. [8] have tailored an algorithm based on the aggregate resources already introduced by Toyoda [16] for solving the MDKP. Finally, Hifi et al. [5] proposed a guided local search-based heuristic in which the trajectories of the solutions were oriented by increasing the cost function with a penalty term; it penalizes bad features of previously visited solutions.

3. A generic approach

In order to make the paper more clearer, we try to summarize the main principle of the algorithm using a generic approach. Note that our algorithm uses some specific parameters associated to the MMKP problem and so, it can be considered as a tailored algorithm using the main lines of the generic approach which is composed by the following steps:

(i) Starting by an initial solution;
(ii) Constructing a neighborhood set in order to improve the current solution, applying a neighborhood-strategy;
(iii) Perturb the search process and construct a new current solution;
(iv) Steps (ii)–(iii) are repeat until a satisfactory solution is reached.

In what follows, we develop a manner for simulating the process illustrated by steps (i)–(iv).

3.1. A starting solution

Simulation of step (*i*): The algorithm starts with a *partial solution* using a greedy procedure. The last procedure works by fixing, at each step, an item until a feasible solution of the problem is obtained; that is a *starting solution*. The obtained solution is reached by considering some choice criterions, for example, the average cost criterion, the marginal cost criterion, etc. Of course, the procedure is a greedy one and the aim is to obtain a feasible solution to the problem in a negligible computing time.

3.2. Defining a neighborhood set

Simulation of step (*ii*): Constructing a neighborhood-solution represents the core of the proposed approach. The neighborhood-solution is used for improving the quality of the

current solution. It consists in finding a good strategy for generating a final solution x_f ($f \geq 1$) localized in the neighborhood-solution. The process applies a series of movements for constructing a series of feasible solutions $x_1, x_2, \ldots, x_n$.

Let consider the following maximization problem:

$$\max_{x \in X}\{f(x)\},$$

where X denotes the domain of x. Define an operator $h(x_t)$ which transforms a current solution x_t into a neighbor solution x_{t+1}; that is a new solution obtained at the t-th iteration using the neighborhood-strategy (operator). The operator $h(.)$ transforms the current solution x_t into the solution x_{t+1} such that $f(x_t) \leq f(x_{t+1})$. The process is stopped if it is incapable to produce a better solution.

The introduced operator can be defined as follows. Let $x_t \in X$ be a feasible solution obtained at the t-th iteration and, suppose that there exists a fixed number p of neighborhood sets for x_t, i.e., $N^1(x_t), \ldots, N^k(x_t), \ldots, N^p(x_t)$. Define the problem P^k corresponding to $N^k(x_t)$, $k = 1, \ldots, p$, as follows:

$$\max_{x \in N^k(x_t)}\{f(x)\},$$

and consider the following two steps:

(a) Let x_t^k, $k = 1, \ldots, p$, denote the best solution localized in P^k with objective value $f(x_t^k)$;
(b) Set $x_{t+1} = \text{argmax}_{y \in \{x_t^1, \ldots, x_t^k, \ldots x_t^p\}}\{f(y)\}$.

3.3. Perturbing the search process

Simulation of step (*iii*): The aim of the procedure is to construct a new starting solution by using a jump from the current space search to a new space search (a diversification is introduced). Several strategies can be applied in order to simulate the jumping principle. For example, (i) a first strategy can degrade the current solution and combines some mechanisms in order to improve the solution at hand; (ii) a second strategy can degrade the current solution and try to improve it using a storage list.

In the generic approach, both strategies constitute a reactive local search using (a) the neighborhood sets in order to search the best local solution and, (b) two different strategies using the *deblocking mechanism* or a *local storage list*. First, the deblocking mechanism tries to release the search process when the obtained solution seems to cycle. Second and last, the local storage list is introduced in order to locate the non desirable solutions, and to forbid any configuration having the same objective value.

4. An algorithm for MMKP

For a combinatorial optimization problem, local search algorithms can be described in terms of several basic components. Among these components, we can (generally) distinguish: (i)

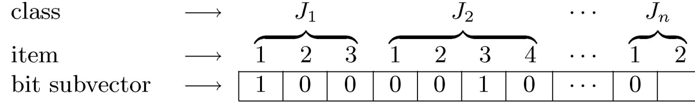

Figure 1. Binary representation of the MMKP solution.

the *combinatorial problem* to solve, (ii) the representation of a *cost function* associated to an instance of the problem and a *neighborhood domain* that defines the possible transitions in the feasible search space and, (iii) the *control strategy* for local moves to perform. Many strategies have been proposed that address the problem of how to overcome local optima. In many cases, non-improving local moves are admitted based on a probabilistic decision (*noising*) or based on the history of the search. The focus of the reactive search framework is on wide spectrum heuristic algorithms for discrete optimization, in which local search is complemented by feedback schemes ("reactive"), that use the past history of the search to increase their efficiency and effectiveness.

4.1. Solution representation for MMKP

Before describing the main principle of the approach, we give a suitable representation scheme and introduce some notations.

Generally, the scheme is a way to represent a solution of MMKP. The standard MMKP binary representation is an obvious choice for MMKP since it represents the underlying 0–1 nonnegative variables (figure 1 shows the vector representation of this solution). A feasible solution is such that $\forall\, k \in \{1, \ldots, m\}$, $\sum_{i=1}^{n} \sum_{j=1}^{r_i} w_{ij}^k\, x_{ij} \leq C^k$ and for each class J_i, we pick one and only one item j, i.e., $x_{ij} = 1$ if the j-th item of the i-th class has been selected, $x_{ij} = 0$ otherwise.

Generally, the binary representation can introduce the unfeasibility in the resulting solutions. Therefore, there are two ways of dealing with unfeasible solutions:

1. to apply a penalty function to penalize the cost function of unfeasible solutions (see Richardson et al. [15]),
2. to design heuristic operators in order to transform the unfeasible solutions into feasible ones (see Beasley and Chu [2]), or to reduce the order of the unfeasibility amount of the obtained solutions.

Penalty techniques allow constraints to be violated; as unfeasibility increases, the cost function is degraded. The second technique consists in separating the cost function into two terms: (i) a term which represents the objective function and (ii) a second one, denoted m^p, which measures the amount of unfeasibility in the solution. Of course, a solution with $m^p = 0$ represents a feasible one for the original problem.

In our study, we apply the second approach of using heuristic operators, because (for the MMKP problem) (i) the objective function is easily computed and (ii) by using a simple procedure, the measure of the unfeasibility of each constructed solution can easily be reduced to zero. We have preferred this approach because an "appropriate" penalty

function is often difficult to determine. In our study, we distinguish two states: the *feasible state* (FS) and the *unfeasible state* (US); FS indicates that the current solution, namely S, does not violate the amount of available constraints, and US if there exist at last one constraint which has been violated for S. The aim is to try to improve FS (or transform US into FS) by applying an *improving greedy* procedure.

In what follows, in order to make the paper self-contained and for increasing clarity of both versions of the proposed algorithm, we recall briefly some parts having already been developed in Hifi et al. [5], which are principally concerned with both the constructive and the complementary procedures.

4.2. An initial solution for MMKP

A *constructive procedure* (CP) and a *complementary* one (CCP), proposed in Hifi et al. [5], construct an initial solution by applying CP and improve it by applying CCP. The first procedure CP operates in a greedy way in order to produce a feasible solution without focalizing on the quality of the obtained solution. CPP, which is a complementary procedure, is applied in order to improve the quality of the initial solution. Herein, we summarize the main principle of both CP and CCP procedures.

4.2.1 CP procedure. CP is a greedy procedure with two phases. The first phase is applied in order to produce a feasible solution of MMKP; the procedure stops if the obtained solution is feasible. The second phase is used when the obtained partial solution is not feasible. In this case, in a greedy way, the procedure considers the most violated constraint and attempts to complete the current partial solution. The last phase is iterated until a feasible solution is obtained (for more details, the reader can refer to Hifi et al. [5]).

4.2.2 CCP procedure. The *complementary CP* approach (CCP), uses an iterative improvement of the starting solution. It applies (i) a swapping strategy of picked items (considered as *old items*) and (ii) a replacement stage which consists of replacing the old item with a new one selected from the same class. Note that each swap is authorized if the obtained solution realizes a FS. By this way, first, the swap is generalized to the remaining items of the same class in order to select the *new item* realizing the *best local objective value* of the current class. Second, the two selected items, say j_i and j_i', of the same class, say J_i, are exchanged in the new solution, where the obtained objective value realizes the better value over all classes. This process is iterated by using a stopping condition (a detailed description of the CCP algorithm can be found in Hifi et al. [5]).

4.3. A reactive local search—RLS

In this section, we describe the first version of the algorithm, called RLS. RLS uses a process in order to improve the solution applying CCP.

The used method is characterized by the “prohibition period” which is determined through feedback (reactive) mechanism during the search. This principle permits us to release the current solution and considers another better solution. The algorithm simulates

this process (as indicated in the generic approach) by considering a two-stage solution generation. The core of the algorithm is mainly based on two strategies: (a) the *degrading strategy* which is applied after improving the current solution by performing some swapping between several items and, (b) the *deblocking strategy* which allows some diversification and permits to change the direction of the search in order to explore some different regions of the search space. Both `Degrade()` and `Deblock()` procedures are summarized as follows.

4.3.1 The degrade procedure. The procedure involves the parameter $S = (S_1, \ldots, S_i, \ldots, S_n)$ denoting the current solution to degrade. The procedure operates as follows:

`Step 1.` Set $J_i \leftarrow GetClass()$, where $GetClass()$ selects an arbitrary class.
`Step 2.` Set $j'_i \leftarrow Exchange(S, J_i, j)$ where (i) j and j'_i are two elements of J_i and, (ii) the exchange between j and j'_i guaranties a feasible solution.
`Step 3.` Repeat steps 1–2 a certain number of times (corresponding to the degrading strategy) and exit with the new solution S.

Note that the procedure at Step 2 uses a simple exchange between items of the same class, i.e., two items are exchanged if the engendered solution is feasible.

4.3.2 The deblock procedure. The main idea of the procedure is based on a double exchange between items of two selected classes. We recall that initially we have a feasible solution S, where $S(i)$ represents the i-th fixed item in the current class, with its objective value *best*.

Now, let i_1 and i_2 be two different classes and, $j_1 \in J_{i1}$ and $j_2 \in J_{i2}$ be two items to be exchanged with two other items of J_{i1} and J_{i2}, respectively. Then, we define the following configuration S', where $S'(i)$ (for $i = 1, \ldots, n$) denotes the index of the fixed item in the current class:

$$S'(i_1, i_2, j_1, j_2) = \begin{cases} S(i) & \text{if } i \notin \{i_1, i_2\} \\ j_1 & \text{if } i = i_1 \\ j_2 & \text{if } i = i_2. \end{cases}$$

The main steps of the deblock procedure, which tries to improve the best current solution, can be described as follows.

1. Set $\mathcal{E} = \{(i_1, i_2)$ such that $i_1, i_2 \in \{1, \ldots, n\},\ i_1 < i_2 \leq n\}$.
2. Choose an element (i_1,i_2) (representing a couple two different classes) of $\mathcal{E}$ and update the set $\mathcal{E}$, i.e., $\mathcal{E} \leftarrow \mathcal{E} \setminus \{(i_1, i_2)\}$.
3. If there exist a couple of elements (j_1, j_2) such that $S'(i_1, i_2, j_1, j_2)$ produces a feasible solution and its objective value $O(S')$ is greater than $Best$, then set $S \leftarrow S'$, $Best \leftarrow O(S')$ and, `exit` with both S and *best*.
4. Repeat Steps 2 and 3 until $\mathcal{E} \neq \emptyset$.

Figure 2 describes the main steps of the algorithm, denoted RLS. The algorithm starts (`line 1`) by applying CP to obtain an initial solution and initializes (`line 2`) the number of

Input: A feasible solution S with objective value $O(S)$.
Output: An improved solution S^* with objective value $O(S^*)$.

```
1. Set S* := S := CP_sol /* CP_sol denotes the solution reached by applying CP */
2. Set p := 0 /* p is the number of times that some solutions can be degraded */
3. While (not StoppingCondition()) do:
4.     S := CCP_S;
5.     If O(S*) < O(S) then S* := S and p := 0;
6.     If (p < Const) then S := Degrade(S) and set p := p + 1;
7.     Else
8.         S:=Deblock(S)
9.         If O(S) > O(S*) then S* := S and p := 0
10.        Else exit with the best current solution;
11. EndWhile
12. Exit with the best solution S* with objective value O(S*).
```

Figure 2. A *reactive local search* using deblocking and degrading strategies: RLS.

times that some solutions can be degraded. Each solution is represented by a configuration recorded in the vector S. $S = (S_1, \ldots, S_n)$ with S_i denotes the item selected in the i-th class. The main loop (`line 3`) starts by performing a *local swapping search* strategy (CCP_S - `line 4`) in order to obtain a first improved solution. The best current solution is updated (`line 5`) if the obtained solution realizes a better objective value compared to the initial one. If the local swap search (CCP_S) is unable to improve the solution (`line 6`), then the *degrading strategy* (the procedure `Degrade()`) is introduced in order to consider another solution. The aim is to change the trajectory of the search which enables a better improvement process. This strategy is repeated for a fixed number of iterations. As described by the `line 8`, if the obtained solution is captured by a local optima, then we try to avoid this phenomenon by using the *deblocking strategy* (`Deblock` procedure). Finally, the local swap search procedure is recalled for improving the current solution. This process is controlled by a fixed number of iterations (`StoppingCondition()`).

The complexity of this algorithm is given as follows. First, RLS starts by calling CP with a complexity of $O(m\ell + n)$, where $\ell = \max\{r_1, \ldots, r_n\}$ (see Hifi et al. [5]). Second, the procedure CCP_S has a complexity of $O(nm\ell)$ and both `Degrade` and `Deblock` procedures have a complexity of $O(m)$. So, the total operations taken by RLS is equal to $MaxIter \times ((m\ell + n) + 2(nm\ell)) \times m$. Finally its worst-case complexity is evaluated to $O(n\ell\, m^2)$.

4.4. A modified RLS algorithm

In this section, we describe a modified version of RLS, called MRLS. The aim of MRLS is to improve the quality of the solutions obtained by RLS. Of course, in our results, we also consider the run time as a critical measure in the sense that we try to generate best solutions without augmenting considerably the computing time. A manner to respect this criteria is to introduce a memory list in order to prevent cycling. Indeed, search procedures based upon local optimization usually require some type of techniques to overcome local optimality.

Input: A feasible solution S with objective value $O(S)$.
Output: An improved feasible solution S^* with objective value $O(S^*)$.

1. Set $S^* := S :=$ CP$_{sol}$;
2. Set $List = \emptyset$
3. $S' \longleftarrow$ CCP$(S, List)$;
4. `Update` (S', S^*);
5. `Insert` $(List, O(S'))$;
6. $S \longleftarrow$ `Degrader`$(S^*, List, p)$,
   /* p denotes the number of times that some solutions can be degraded */
7. Repeat steps 3-6 until a satisfactory solution S^* (with objective value $O(S^*)$) is obtained.

Figure 3. A *modified reactive local search* algorithm using a memory list: MRLS.

We can remark that after a move is executed, using RLS, one checks whether the current configuration has already been found during the search process and reacts accordingly. The later reactive mechanism is not sufficient to guarantee that the search trajectory is not confined in a limited region of the search space, for instance, when the current solution saturates relatively all constraints. For this reason, the robustness of the approach requires other "escape" mechanism called herein the *memory storage*, applied when several configurations realize the same objective value. In our study, we propose to replace the `Deblock()` strategy by a memory list. Limited computational results showed that combining both `Deblock()` and memory list produce, generally, equivalent solutions but the computational time becomes superior.

Figure 3 describes the main steps of the MRLS. It starts (`lines 1 and 2`) by applying CP to obtain a starting solution denoted $S^* := S$, with objective value $O(S^*)$ and, initializing the memory list to empty set. The algorithm (`line 3`) performs some local swapping search (using CCP(S, $List$)) to improve the current solution which does not belong to tabu set in the memory $List$. Next (`line 4`), the obtained solution S' is compared to the recorded objective value; the current obtained solution replaces the older solution, noted S^*, if its objective value is better than $O(S^*)$. The solution S' (`line 5`) is introduced in the tabu list in order to limit cycling phenomenon. Moreover, the algorithm applies the degrade procedure at `line 6` in order to construct a new solution S. Note that the algorithm performs a maximum of p degradations in order to construct the later solution. This process is repeated until a maximum number of iterations is performed.

Note that MRLS has the same complexity than RLS in the sense that it uses the same procedures except the fact that instead of the `Deblock()` procedure, it uses a memory list.

5. Computational results

The purpose of this section is twofold: (i) to show how to determine a good trade-off between the quality of the obtained solution, the size of the used memory $List$, the accepted cycling length and the number of times of degrading solutions and, (ii) to evaluate the performance of both versions of the algorithm compared to the results obtained by Hifi et al.'s [5] algorithm (referred to herein as *HMS*). The obtained results are also compared to those obtained when running one hour the `Cplex Solver v.9` on the same set of instances.

Table 1 Test problem details.

#Inst.	n	r_i	m	$\sum_{i=1}^{n} r_i$	#Inst.	n	r_i	m	$\sum_{i=1}^{n} r_i$	#Inst.	n	r_i	ms	$\sum_{i=1}^{n} r_i$
I01	5	5	5	25	Ins01	50	10	10	500	Ins13	100	30	10	3000
I02	10	5	5	50	Ins02	50	10	10	500	Ins14	150	30	10	4500
I03	15	10	10	150	Ins03	60	10	10	600	Ins15	180	30	10	5400
I04	20	10	10	200	Ins04	70	10	10	700	Ins16	200	30	10	6000
I05	25	10	10	250	Ins05	75	10	10	750	Ins17	250	30	10	7500
I06	30	10	10	300	Ins06	75	10	10	750	Ins18	280	20	10	5600
I07	100	10	10	1000	Ins07	80	10	10	800	Ins19	300	20	10	6000
I08	150	10	10	1500	Ins08	80	10	10	800	Ins20	350	20	10	7000
I09	200	10	10	2000	Ins09	80	10	10	800					
I10	250	10	10	2500	Ins10	90	10	10	900					
I11	300	10	10	3000	Ins11	90	10	10	900					
I12	350	10	10	3500	Ins12	100	10	10	1000					
I13	400	10	10	4000										

Our algorithms were coded in `C++` and all algorithms are tested on an `Ultra-Sparc10` (250 Mhz and with 128 Mb of RAM).

5.1. *Problem details*

The problems we considered are summarized in Table 1. We tested a total of 33 instances corresponding to two groups: (i) the first group contains existing instances (noted `I01`, ... , `I13`) and (ii) the second group is composed of randomly generated problem instances (noted `Ins1`, ... , `Ins20`). The first group corresponds to the instances tested by Khan et al. [8] containing six small and seven hard instances. The second group represents a variant of random problem instances (varying from medium to large size instances) which are generated following the scheme used in [8]. We have made these instances publicly available from http://www.laria.u-picardie.fr/hifi/OR-Benchmark, hoping to aid further development of exact and approximate algorithms for the MMKP.

5.2. *Behavior analysis: The literature instances*

Generally, when using approximate algorithms to solve optimization problems, it is well-known that different parameter settings for the approach lead to results of variable quality. Herein, both versions of the approach (RLS and MRLS) involve several decisions. For RLS, we have the parameter p representing the number of times that the current solution can be degraded and, the number of iterations *MaxIter* to do. For MRLS, we add to the parameter p, the length of the used tabu list. Of course, a different adjustment of method's parameters would lead a hight percentage of good solutions. But this better adjustment

Table 2 The behaviour of RLS when varying the number of iterations *MaxIter* and by fixing p to 5.

# Iterations	AV. T	# Exact/Best solutions (%)
$5n$	2.42	7 (0.54)
$10n$	6.14	7 (0.54)
$15n$	8.04	8 (0.61)
$50n$	21.95	12 (0.92)

would sometimes lead to heavier execution time requirements. The set of values chosen in our experiment represents a satisfactory trade-off between solution quality and running time.

First, in order to find the right value of the maximum number of iterations used by the algorithm, we have introduced a variation of *MaxIter* in the discrete interval $\{5n, 15n, 20n, 50n\}$. Moreover, when the number of the maximum iterations is greater or equal to $15n$, the search process is stopped if the current solution is not improved after n degradations. These tests are made by fixing the parameter p, representing the number of times of degrading a current solution, to 5 (below, we shall discuss the choice of the value associated to p). Limited computational results showed that a hight value of *MaxIter* gives a better solution, but the computational times increase. As shown in Table 2, we can observe that the percentage of optimal (best) solutions varies between 0.54% and 0.92%. Note that the better solutions are obtained when fixing *MaxIter* to $50n$ with a largest average computational time (Line 3, Column 3). In what follows, we maintain the last value for the maximum number of iterations.

Second, we analyze the behavior of RLS when varying the parameter representing the maximum number of degrading a current solution, denoted p. Table 3 reports the quality of the obtained results when p is varied in the discrete interval $\{5, 15, 20\}$. The same table shows that he algorithm produces better results for $p = 5$ and if the value of p is large, then the used diversification is less important (in the sense that the approach is unable to provide better solutions). However, we think that for the largest value the algorithm explores a large space and so, the reactive search is not able to locate a good direction in order to improve some visited solutions. From Table 3, we can conclude that an intermediate value for p maintains the hight quality of the solutions.

Third and finally, we consider the tabu list used in the second version of the algorithm, i.e. MRLS. We recall that the tabu list replaces the unblocking strategy. In our study, we have considered that the length of the tabu list varies dynamically. Indeed, if n^* denotes the

Table 3 The behaviour of RLS when varying p, the number of times a solution must be degraded.

Varying p	AV. T	# Exact/Best solutions (%)
5	21.95	12 (0.92)
15	27.77	10 (0.77)
20	33.65	7 (0.54)

Table 4 The behaviour of MRLS when varying dynamically the length of the memory list.

Interval list	AV. T	# Exact/Best solutions (%)
$[n, n+10]$	21.95	12 (0.92)
$[2n, 2n+10]$	42.47	10 (0.77)
$[3n, 3n+10]$	46.74	8 (0.62)

number of the different classes, then the length is automatically taken in an integer interval. The change of tabu list is performed after 50 iterations without improving the best current solution. In order to find a right interval variation associated to the length of the tabu list, we have compared several intervals. Table 4 displays the results obtained by MRLS when varying the length of the tabu list (herein, we have reported three significant intervals).

We can remark that the better results are obtained for the interval $[2n, 2n+10]$. In this case, MRLS reaches all optimal/best solutions and consuming an average time equal to 14.52 seconds. The same table shows that if the length of the tabu list is small or large, then the used storage is less efficient. Indeed, we observe that MRLS degrades the quality of the results. We think that for the small interval, the tabu list is not sufficiently able to detect the major cycling and, for the largest one, the list saves several configurations which do not permit to well explore the space search (using the local swap procedure).

5.3. Performance of RLS and MRLS

In this section, we first compare the performance of RLS and MRLS to that of Hifi et al.'s [5] algorithm, referred to herein as HMS and, we give the detailed results of both algorithms for the first group of instances, i.e., the instances of the literature noted `I01-I13`. Second, we perform a comparative study between the proposed algorithms and the `Cplex solver v.9` on both groups of instances (the second group contains the instances `Ins01-Ins20` representing a variant of problems varying from medium to large ones).

5.3.1 The first group of instances. Table 5 evaluates the performance of the proposed algorithms (RLS and MRLS) compared to HMS on the first group of instances. Column 2, labeled *Opt*/*Best* contains either the optimal solution, if it is known, or the best feasible solution obtained up to now. (Whenever a best feasible solution is used, the instance is marked with a $*$ sign). Columns 3 and 4 display the solutions obtained by HMS and its run time while Columns 5 to 8 display the solutions yielded by the two versions of MRLS (denoted MRLS^a and MRLS^b) and the used run times. Recall that the implementation of MRLS involves the set up of several parameters and in particular the parameter relied to the maximum number of iterations. For the first implementation of MRLS, denoted MRLS^a, we fixed the maximum number of iterations to $15n$ (favoring the run time) and to $50n$ for the second version, denoted MRLS^b (favoring the quality of the solutions). These strategies seem, among the strategies we explored, to be a good compromise in terms of solution quality and run time. It is clear that applying MRLS with a large number of iterations can produce better solutions but it requires large computational time.

Table 5 A summary results of *HMS, RLS and MRLS* algorithms. The symbol $*$ means that the optimal solution is not known.

						MRLS algorithm			
Inst.	*Opt/Best*	HMS	T	RLS	T	MRLS^a	T^a	MRLS^b	T^b
I01	173	173	0.04	173	0.05	173	0.56	173	0.59
I02	364	356	0.04	364	0.07	364	0.79	364	0.81
I03	1602	1553	0.08	1595°	0.20	1602	1.99	1602	2.01
I04	3597	3502	0.09	3564°	0.22	3569	2.28	3597	2.30
I05	3905.70	3868.22	0.15	3905.90°	0.19	3945.70	1.93	3905.70	1.94
I06	4799.30	4799.30	0.21	4799.30	0.23	4799.30	2.36	4799.30	2.37
I07	24587*	23983	1.50	24121°	1.20	24159	11.66	24587	36.58
I08	36877*	36007	2.17	36110	2.37	36401	14.93	36877	37.00
I09	49167*	48048	5.50	48291°	4.33	48367	20.03	49167	25.10
I10	61437*	60176	7.47	60291°	6.65	60475	25.45	61437	47.00
I11	73773*	72003	13.35	72283°	9.52	72558	30.27	73773	41.45
I12	86071*	84160	22.41	84446°	12.69	84707	36.32	86069	42.08
I13	98429*	96103	31.64	96580°	16.73	96834	41.20	98429	160.41

The study of Table 5 shows that RLS gives an acceptable improvement of results given by *HMS* (Column 5 marked with the symbol $\circ$) within shortest average computational time. We also remark that either implementation of MRLS reach better solutions than both HMS and RLS. For this group of instances (compared to RLS results), MRLS^a improves all instances (both algorithms produce the optimal solution for three instances) while MRLS^b (compared to MRLS^a) improves the solutions of all instances (for five instances, both versions of MRLS give the optimal solutions). Evidently, these improvements occur at the cost of a larger computational time. Moreover, MRLS^b needs more average run time and reaches better solutions. However, the improvement of the solution quality warrants the additional (reasonable) run time.

5.3.2 A comparative study on both groups of instances. In this part, we compare the results produced by both MRLS^b and the `Cplex solver v.9` on both groups representing 33 problem instances. This comparison is performed by setting the run time limit of the `Cplex solver` to one hour. Table 6 evaluates the performance of both algorithms on both groups of instances. Column 3 shows the optimal solution (or the best solution; in this case, the instance is marked with an $*$ sign) of the instance. Column 3 contains the *best integer feasible solution Cplex$_{IFS}$* produced by the `Cplex solver`. Column 4 tallies the solutions given by MRLS^b while column 5 displays the computational time that needs MRLS^b for producing the final solutions. All entries in italic (Columns 3 and 4) indicate which algorithm reaches the better solution for the considered instance.

For the first group of instances, we can remark that MRLS^b produces five better solutions out of seven (for the first six instances, both algorithms produce the optimal solutions). For the second group of instances, MRLS^b performs (on average) better than the Cplex solver,

Table 6 Computational results of both `Cplex Solver` and MRLS algorithm. The symbol $*$ means that the optimal solution is not known.

Inst.	Opt/Best	$Cplex_{IS}$	$MRLS^b$	T^b
I01	173	173	173	0.59
I02	364	364	364	0.81
I03	1602	1602	1602	2.01
I04	3597	3597	3597	2.30
I05	3905.70	3905.70	3905.70	1.94
I06	4799.30	4799.30	4799.30	2.37
I07	24587*	24584	*24587*	36.58
I08	36877*	36869	*36877*	37.00
I09	49167*	49155	*49167*	25.10
I10	61446*	*61446*	61437	47.00
I11	73773*	73759	*73773*	41.45
I12	86071*	*86071*	86069	42.08
I13	98429*	98418	*98429*	160.41
Ins01	10714*	10709	*10714*	10.27
Ins02	13598*	13597	*13598*	76.00
Ins03	10943*	10934	*10943*	58.00
Ins04	14429*	14422	*14429*	7.69
Ins05	17053*	17041	*17053*	42.00
Ins06	16823*	16815	*16823*	50.00
Ins07	16423*	16407	*16423*	65.00
Ins08	17506*	17484	*17506*	26.78
Ins09	17754*	17747	*17754*	51.23
Ins10	19314*	19285	*19314*	32.16
Ins11	19431*	19424	*19431*	110.98
Ins12	21730*	21725	*21730*	23.39
Ins13	21569*	21569	21569	18.00
Ins14	32869*	32866	*32869*	72.00
Ins15	39154*	*39154*	39148	63.00
Ins16	43357*	*43357*	43354	194.00
Ins17	54349*	54349	54349	30.00
Ins18	60456*	60455	*60456*	201.00
Ins19	64921*	64919	*64921*	45.00
Ins20	75603*	75603	75603	47.00

since it produces 16 better solutions out of 17 (in three cases, both algorithms reach the same feasible solution).

Finally, it is noteworthy that (i) if fast solutions are needed, RLS should be used, (ii) if high quality solutions are preferred to speed, $MRLS^b$ should be chosen and, (iii) if

intermediate solutions within reasonable computing time are needed, MRLS^a can be used or using MRLS by varying the maximum number of iterations.

6. Conclusion

We have solved the multiple-choice multi-dimensional knapsack problem using two approximate algorithms: a simple reactive local search and a modified one. The first approach is mainly based upon a local search and combining degrading and deblocking procedures. The degrading strategy is used in order to diversify the search and the deblocking one is introduced for escaping to local optima and trying to improve the quality of the solutions. The second approach introduces a memory list which replaces the deblocking strategy. The aim is to record cycling solutions in order to forbid repetition of configurations in the solutions. Computational results show that the first approach yields good solutions within a very short computing time. The second approach yields high quality solutions, reaching the optimal/best for several instances, within a reasonable run time.

Acknowledgments

The authors thank the anonymous referees and the Guest Editor for their helpful comments and pertinent suggestions which contributed to the improvement of the presentation and the contents of this paper.

References

1. E. Balas and E. Zemel, “An algorithm for large zero-one knapsack problem,” Operations Research, vol. 28, pp. 1130–1154, 1980.
2. P. Chu and J.E. Beasley, “A genetic algorithm for the multidimensional knapsack problem,” Journal of Heuristics, vol. 4, pp. 63–86, 1998.
3. J.E. Beasley and P.C. Chu, “A genetic algorithm for the set covering problem,” Europ. J. Opl. Res, vol. 94, pp. 392–404, 1996.
4. D. Fayard and G. Plateau, “An algorithm for the solution of the 0-1 knapsack problem,” *Computing,* vol. 28, pp. 269–287, 1982.
5. M. Hifi, M. Michrafy and A. Sbihi, “Heuristic algorithms for the multiple-choice multidimensional knapsack problem,” Journal of the Operational Research Society, vol. 55, pp. 1323–1332, 2004.
6. H. Kellerer, U. Pferschy, and D. Pisinger, Knapsack Problems, Springer, 2003.
7. S. Khan, K.F. Li, E.G. Manning, and MD. M. Akbar, “Solving the knapsack problem for adaptive multimedia systems,” Studia Informatica, an International Journal, Special Issue on Cutting, Packing and Knapsacking Problems, vol. 2, no. 1, pp. 154–174, 2002.
8. S. Khan, “Quality adaptation in a multi-session adaptive multimedia system: Model and architecture,” PhD Thesis, Department of Electronical and Computer Engineering, Uiversity of Victoria, May 1998.
9. H. Luss, “Minmax resource allocation problems: Optimization and parametric analysis,” European Journal of Operational Research, vol. 60, pp. 76–86, 1992.
10. S. Martello, D. Pisinger, and P. Toth, “Dynamic programming and strong bounds for the 0-1 knapsack problem,” Management Science, vol. 45, pp. 414–424, 1999.
11. M. Moser, D.P. Jokanović, and N. Shiratori, “An algorithm for the multidimesional multiple-choice knapsack problem,” IEECE Transactions on Fundamentals of Electronics, vol. 80, no. 3, pp. 582–589, 1997.
12. 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.

13. D. Pisinger, “A minimal algorithm for the 0-1 knapsack problem,” Operations Research, vol. 45, pp. 758–767, 1997.
14. D. Pisinger, “A minimal algorithm for the Multiple-choice Knapsack Problem,” European Journal of Operational Research, vol. 83, pp. 394–410, 1995.
15. J. Richardson, M. Palmer, G. Liepins, and M. Hilliard, “Some guidelines for genetic algorithms with penalty functions,” in Proceedings of the Third International Conference on Genetic Algorithms, Schaffer J (Eds.), Maurgan Kaufmann, 1989 pp. 191–197.
16. Y. Toyoda, “A simplified algorithm for obtaining approximate solution to zero-one programming problems,” Management Science, vol. 21, pp. 1417–1427, 1975.