Journal of the Operational Research Society (2004) 55, 1323–1332
www.palgrave-journals.com/jors

Heuristic algorithms for the multiple-choice multidimensional knapsack problem

M Hifi[1,2]*, M Michrafy[2] and A Sbihi[1]

[1]*LaRIA, UPJV, Amiens, France; and* [2]*CERMSEM-CNRS UMR 8095, Universite de Paris 1, Paris, France*

In this paper, we propose several heuristics for approximately solving the multiple-choice multidimensional knapsack problem (noted MMKP), an NP-Hard combinatorial optimization problem. The first algorithm is a constructive approach used especially for constructing an initial feasible solution for the problem. The second approach is applied in order to improve the quality of the initial solution. Finally, we introduce the main algorithm, which starts by applying the first approach and tries to produce a better solution to the MMKP. The last approach can be viewed as a two-stage procedure: (i) the first stage is applied in order to penalize a chosen feasible solution and, (ii) the second stage is used in order to normalize and to improve the solution given by the firs stage. The performance of the proposed approaches has been evaluated based problem instances extracted from the literature. Encouraging results have been obtained.
Journal of the Operational Research Society (2004) **55,** 1323–1332. doi:10.1057/palgrave.jors.2601796
Published online 14 July 2004

Keywords: combinatorial optimization; guided local search; heuristic, knapsack

Introduction

The multiple-choice multidimensional knapsack problem (MMKP) is a more complex variant of the 0–1 knapsack problem, an NP-Hard problem. Due to its high computational complexity, algorithms for the exact solution of the MMKP are not suitable for most real-time decision-making applications, such as quality adaptation and admission control for interactive multimedia systems,[1] or service level agreement management in telecommunication networks.[2] In the MMKP, we are given n classes J_i of items, where each class J_i, $i=1, \ldots, n$, has r_i items. Each item j, $j=1, \ldots, r_i$, of class J_i has the non-negative profit value v_{ij}, and requires resources given by the weight vector $W_{ij}=(w_{ij}^1, w_{ij}^2, \ldots, w_{ij}^m)$ where each weight component w_{ij}^k, $k=1, \ldots, m$ also is a non-negative value. The amounts of available resources are given by a vector $C=(C^1, C^2, \ldots, C^m)$. The aim of the MMKP is to pick exactly one item from each class in order to maximize the total profit value of the pick, subject to resource constraints. Formally, the MMKP can be stated as follows:

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

The variable x_{ij} is either equal to 0, implying item j of the ith class J_i is not picked, or equal to 1 implying item j of the ith J_i class is picked.

In this paper, we propose several algorithms for the MMKP. The first two algorithms can be considered as constructive and complementary solution approaches. The third algorithm is mainly based upon a *guided local search* (GLS) method (for more details, the reader can be referred to Voudouris and Tsang[3,4] which has its origin in constraint satisfaction applications. GLS has proven to be a very powerful approach for solving several hard combinatorial optimization problems. It uses memory to guide the search to promising regions of the solution space. This is performed by increasing the cost function with a penalty term that penalizes bad features of previously visited solutions. In this work, we introduce a new principle based on the following points: (a) starting with a lower bound obtained by a fast greedy procedure, (b) improving the quality of the initial solution using a complementary procedure and (c) searching the best feasible solution over a set of neighbourhoods. The main idea consists in choosing a penalty strategy to construct a better solution on the space of the feasible solutions.

The remaining of the paper is organized as follows. First, we present a brief reference of some sequential exact and approximate algorithms for knapsack problem variants. Second, we present the concept of GLS and the main principle of the proposed algorithms. Third, we propose two constructive procedures used especially for providing an initial starting solution to the problem. Fourth, we then present a derived algorithm using (i) a penalty strategy and (ii) a normalized solution. Fifth and last, the paper is concluded with experimental results.

*Correspondence: M Hifi, CERMSEM, Universite de Paris 1, Pantheon-Sorbonne, 106-112 Boulevard de l'Hôpital, Paris Cedex 13, 75647, France.
E-mail: hifi@univ-paris1.fr

Literature survey

Most of the researches on knapsack problems deal with the much simpler constraint version ($m=1$ and $n=1$). For the single constraint case the problem is not strongly NP-Hard and effective approximation algorithms have been developed for obtaining near-optimal solutions. A good review of the single knapsack problem and its associated exact and approximate algorithms is given by Martello and Toth.[5] Below we review the literature for the knapsack problem variants. We will particularly discuss the multidimensional knapsack problem (MDKP), the multiple-choice knapsack problem (MCKP), and the MMKP.

Many variants of knapsack problems, which comprise an important class of combinatorial optimization, have been thoroughly studied in the last few decades (see Chu and Beasley[6] and Martello and Toth[7]). There are two types of solution methods: *exact algorithms* capable to produce the optimal solutions for some problem instances within a reasonable computational time, and *approximate procedures or heuristics* capable to produce 'good' (near-optimal) solutions within small computational time.

Most exact algorithms for solving the knapsack problem (KP) variants are mainly based on (i) branch-and-bound search using depth-first search strategy (see Balas and Zemel,[8] Fayard and Plateau[9] and Martello and Toth[5,10]), (ii) dynamic programming techniques (see Pisinger[11]), and (iii) hybrid algorithms combining dynamic programming and branch-and-bound procedures (see Martello *et al*[12]). The MDKP is a generalization of the classical binary knapsack problem for multiple resource constraints. For more details see Chu and Beasley,[6] Freville and Plateau[13] and Shih.[14] Another variant of the knapsack problem is the MCKP, where the picking criterion for items is more restricted. For the later variant of the knapsack problem there are one or more disjoint classes of items, for more details, one can refer to Nauss.[15] Finally, the MMKP can be considered as a more generalization of the MDKP and MCKP variants of the binary knapsack problem (0–1 KP). Most algorithms for optimal solutions of knapsack problem variants are also based upon branch-and-bound procedures (see Nauss,[15] Khan[16] and Pisinger[17]).

A greedy algorithm has been proposed for approximately solving the knapsack problems (see Martello and Toth[7]). For the classical binary knapsack problem, the approach is composed of two stages: (i) sort the items in decreasing order of value–weight ratio and (ii) pick as many items as possible from the left of the ordered list until the resource constraint is violated. By using the same principle for the MDKP, Toyoda[18] used the aggregate resources consumption. The solution of the MDKP needs iterative picking of items until the resource constraint is violated. Shih[14] presented a branch-and-bound algorithm for MDKP. In this method, an upper bound was obtained by computing the objective function value associated with the optimal fractional solution algorithm (see Dantzig[19]) for the m single constraint knapsack problems and selecting the minimum objective function value among those as the upper bound. In the recent past, great success has been achieved via the application of local search techniques and metaheuristics to MDKP. Most popular has been tabu search, genetic algorithms, simulated annealing and hybrid algorithms (for more details the reader can refer to Chu and Beasley[6]).

An approximate algorithm has been proposed by Moser *et al*[20] for the solution of the MMKP. The algorithm uses the concept of graceful degradation from the most valuable items based on Lagrange multipliers. Finally, Khan *et al*[21] proposed an algorithm based mainly on the aggregate resources already used by Toyoda[18] for solving the MDKP. The method works as follows: (i) it starts with finding an initial feasible solution, (ii) it uses Toyoda's concept of aggregate resources for selecting items to pick, and (iii) it uses iterative improvement of the solution by using some exchanges of picked items.

Solution approaches of the MMKP

A modified GLS algorithm

The GLS algorithm is a recent approach, considered as a metaheuristic, that has proven to be effective on a wide range of hard combinatorial optimization problems. GLS has been first applied by Voudouris and Tsang[3,4] for solving constraint satisfaction problems. It can be considered as a tabu search (see Hansen,[22] Glover[23] and Glover and Laguna[24]), since it uses a memory to control the search processes in a manner similar to tabu search. However, the definition is simpler and more compact. GLS has proved to be an effective approach for non-trivial problems such as the travelling salesman problem,[4,25] quadratic assignment problem,[25] resource allocation,[26] vehicle routing problem,[27] and bin-packing problem.[28] The guided local search moves out of a local maximum/minimum by penalizing particular solution features that it considers should not occur in a near-optimal solution. It defines a modified objective function, augmented with a set of penalty parameters on these features. The usual local search method is then used to improve the augmented objective function. The cycle of local search and penalty parameter update can be repeated as often as required.

In our study, we propose a variant of the GLS algorithm which mainly consists in operating some penalization to the search process in order to escape local optima. First, the *operated penalty* is almost controlled by a fixed depth parameter which initiates several items' profits to penalize, and a parameter, that controls the diameter of the space search. This approach also uses some penalty coefficients chosen with links to the size of the problem. This is in order to operate randomly the penalty to any current solution which indicates that this one is blocked into a gap of very

attractive local optima. The aim of the used penalization is twofold: (i) to release the current solution and (ii) to modify the search trajectory. The later points are introduced in order to make some diversifications and to obtain a 'good' configuration solution.

GLS for the MMKP

Solution representation

Before describing the proposed approaches, we give a suitable representation scheme and introduce some notations.

Let J be a set of items divided into n disjoint classes such that j, $j \in J_i$, has a profit value v_{ij} and a weight vector $W_{ij} = (w_{ij}^1, \ldots, w_{ij}^m)$, and let $C = (C^1, \ldots, C^m)$ be a capacity vector for the multidimensional knapsack. The multidimensional knapsack is subject to multiple-choice constraints which may be formulated as: $\forall i \in \{1, \ldots, n\}$, $\sum_{j=1}^{r_i} x_{ij} = 1$.

The aim is to determine a subset of items such that the sum of their values is maximum without exceeding the m capacity constraints. Generally, the scheme is a way to represent a solution of MMKP. The standard MMKP 0–1 binary representation is an obvious choice for MMKP since it represents the underlying 0–1 non-negative variables (Figure 1 shows the vector representation of an eventual solution).

A feasible solution is such that $\forall k \in \{1, \ldots, m\}$, $\sum_{i=1}^{n} \sum_{j=1}^{r_i} w_{ij}^k x_{ij} \leqslant C^k$ and for each class J_i, we pick one and only one item j, i.e, $x_{ij} = 1$ if the jth item j of the ith class J_i has been selected, otherwise $x_{ij} = 0$.

In what follows, we distinguish the following states: *feasible state* (FS) and *unfeasible state* (US); FS indicates that the current solution, namely S, does not violate the amount of available constraints, and US indicates if there exists at least a violated constraint on S. The goal is to produce an improved FS (or to transform US to FS) by applying a *swapping local search*.

An initial solution for the MMKP

The initial feasible solution is obtained by applying a *constructive procedure*, noted CP. CP is a greedy procedure with two phases: a DROP phase and an ADD phase. This is to generate a feasible solution by considering the FS process. It starts by computing the pseudo-utility ratio $u_{ij} = v_{ij} / \langle C, W_{ij} \rangle$, $j \in \{1, \ldots, r_i\}$ of each item j belonging to each class J_i, where $\langle .,. \rangle$ is the scalar product in $\mathbb{R}^m$. Then it selects the item j from each class J_i, $i \in \{1, \ldots, n\}$, realizing the most valuable u_{ij}. If the obtained solution is an FS, then CP terminates, otherwise (DROP phase) it considers as the most violated constraint C^{k_0}. With respect to C^{k_0}, it selects the class J_{i_0} corresponding to the fixed item j_{i_0} having the largest weight $w_{i_0 j_{i_0}}^{k_0}$ all over the fixed items and regarding the most violated constraint C^{k_0}. This item (ADD phase) is then swapped with another selected item j from the same class J_{i_0}, and the procedure controls the feasibility of the state. If the new obtained solution is a US, it selects the lightest item. j'_{i_0} of the current class J_{i_0} which in turn is considered as the new selected item. This process is iterated until an FS or the smallest unfeasibility amount for the obtained solution is obtained. CP approach may be described by the steps of Box 1.

We can show that CP has a complexity of $O(\max\{\theta m, n\})$ where $\theta = \max\{r_1, \ldots, r_n\}$. Indeed, on the one hand, in the main step (the loop while) one takes m operations to obtain

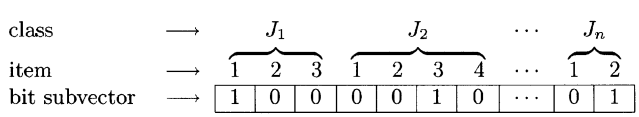

Figure 1 Binary representation of the MMKP solution.

Box 1 The constructive procedure for determining an initial feasible solution: CP

Input: An instance of the MMKP.
Output: A feasible solution S with value $O(S)$.

Initialization.
  For $i = 1, \ldots, n$, set $u_{iji} = \max\{u_{ij}, j = 1, \ldots, r_i\}$;
    $S_i \leftarrow j_i$;
    Set $\phi[i] = j_i$; $x_{i\phi[i]} = 1$;
    Set $R^k = \sum_{i=1}^{n} \omega_{i\phi[i]}^k$, $\forall k = 1, \ldots, m$;
      /* R^k: the accumulated resources for constraint k */
  EndFor;
  $S = (S_1, \ldots, S_n)$;
Main.
  While ($R^k > C^k$, for $k = 1, \ldots, m$) /*DROP phase*/
    $k_0 \leftarrow \underset{1 \leqslant k \leqslant m}{\text{argmax}} \{R^k\}$;
    $i_0 \leftarrow \underset{1 \leqslant i \leqslant n}{\text{argmax}} \left\{\omega_{ij_i}^{k_0}\right\}$;
    $\phi[i_0] = j_{i_0}$; $x_{i_0 \phi[i_0]} = 0$;
    $R^k = R^k - \omega_{i_0 \phi[i_0]}^k$ for $k = 1, \ldots, m$;
    For $j = 1, \ldots, r_{i_0}$ /* ADD phase */
      If ($\exists j \neq j_{i_0}$ and $R^k + \omega_{i_0 j}^k < C^k$, for $k = 1, \ldots, m$) then
        $x_{i_0 j} = 1$;
        $j_{i_0} = j$;
        $\phi[i_0] = j_{i_0}$;
        $R^k = R^k + \omega_{i_0 \phi_{[i_0]}}^k$ for $k = 1, \ldots, m$;
        $S = (\phi[i_0]; \phi[i], \forall i \neq i_0, i = 1, \ldots, n)$ is a feasible solution;
        Exit with the S vector;
      EndIf;
    EndFor;
    $j'_{i_0} \leftarrow \arg min \{\omega_{i_0 j}^{k_0}\}$; /* if the obtained solution is not feasible */
    $j_{i_0} = j'$; $\phi[i_0] = j_{i_0}$; $x_{i_0 \phi[i_0]} = 1$
  EndWhile;
  Return S with value $O(S)$.

k_0 and n operations to obtain i_0. The procedure takes $m\theta$ operations to build a new configuration, where the number of operations consumed for updating the consumed resources is bounded at worst by $(m+1)\theta$ operations. Hence, the complexity of CP is then evaluated at worst to $O(m\theta+n+\theta)$, which is equivalent to $O(m\theta+n)$, which in turn, is also equivalent to $O(\max\{m\theta, n\})$.

The unfeasibility case

In this section, we may discuss the unfeasibility case of the solutions generated by CP, those of US state. In this case, we try to reduce the unfeasibility amount of the solution by running a procedure that uses the local swapping strategy between two items, we say j and j' belonging to the same class J_i. First, we define the ith resource consumption for the kth constraint as $R^k(j_i) = w^k_{iji} + \sum_{(i'\neq i,i'=1)}^{n} \sum_{j=1}^{r_{i'}} w^k_{i'j}$. The best local swap for an item j_i of a class J_i is the one which satisfies a decision criteria (of course, we can define other decision criterion) which realizes the minimum of the following ratio:

$$D_{j_i} = \frac{\sum_{k=1}^{m} R^k(j_i)}{\sum_{k=1}^{m} C^k}$$

This swapping is operated in two stages:

1. Computing the ratios D_{j_i}:
   First, we apply the swapping strategy for the item realizing the smallest value of the decision criteria that we have defined before. More precisely, for each class J_i, $i=1,\ldots,n$: (a) we apply the local swapping strategy between two items, (b) each item for j_i swapped with another from the same class, we compute the ratio D_{j_i}, (c) next, we record the smallest value of the ratio $D_{j_i^{\min}}$ and the index $j_i^{\min}$ of the item which corresponds to the later ratio.
2. Selection of the best ratio:
   From all the computed ratios recorded with their corresponding items, we select the class $J_{i_{\min}}$ that realizes the smallest value over all $D_{j_i^{\min}}$, $i=1, \ldots, n$. Next, we apply the local swapping strategy in the considered class $J_{i_{\min}}$. This phase terminates by updating the consumed resources after operating the global swap between $J_{i_{\min}}$ and $j_{i_{\min}}^{\min}$, which ensures an FS state or less unfeasibility amount of the obtained US solution.

This process is iterated for a fixed number of iterations *MaxIter*, and for each iteration we check the feasibility state of the obtained solution. In the case of an obtained FS state, the process ends and the obtained solution is feasible.

As soon as the process reaches the maximum number of iterations and the obtained solution is of US state, we say that the process is unable to generate a feasible solution. Of course, one also can apply the procedures developed in the following sections used especially to try to transform an US state into a FS one.

In what follows, we describe a complementary local search procedure in order to improve the quality of the solution generated by CP.

A complementary local search

The *complementary CP* approach (CCP), uses an iterative improvement of the initial feasible 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 solution 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 solution value realizes the better solution value over all classes. This process is iterated by using a stopping condition. A detailed description of the CCP algorithm is given in Box 2.

Box 2 A complementary feasible solution: the CCP approach

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

Step 1.
  set $S=(S_1,\ldots,S_n) \leftarrow CP()$;
  set $S^* \leftarrow S$;
Step 2.
  While not StoppingCondition() do
    Forall i in $\{1, \ldots, n\}$ do
      $j_i' \leftarrow$ LocalSwapSearch(j_i, J_i)/*j_i (j_i') denotes the old (new) item of the class J_i */ /* and the exchange between j_i and j_i' is authorized */
      $S_i \leftarrow j_i'$
      $S \leftarrow (S_1,\ldots,j_i',\ldots,S_n)$;
      If $O(S_1,\ldots,j_i',\ldots,S_n) > O(S^*)$ then
      $S^* \leftarrow (S_1,\ldots,j_i',\ldots,S_n)$;
      EndIf
  EndWhile
return S^* with value $O(S^*)$;

First of all, we detail the principle of the LocalSwapSearch() procedure that CCP applies in order to improve the solution generated by CP.

(A) In the begining, the LocalSwapSearch() procedure initializes the best element to swap:
  (A.1) *value* $\leftarrow v_{iS_i}$, where v_{iS_i} denotes the profit of the old fixed item in the ith class J_i to be swapped;
  (A.2) $s_i \leftarrow S_i$, where s_i is a candidate item in J_i to be swapped;

(B) Next, it operates the exchange which may be authorized:

(B.1) For ($j=1, \ldots, r_i$ and $j \neq S_i$) do
If ($v_{ij} > value$ and $R^k - w^k_{iS_i} + w^k_{ij} \leqslant C^k$, $\forall k = 1, \ldots, m$) then
Set $value \leftarrow v_{ij}$;
Set $s_i \leftarrow j$;

B.2) Return s_i as the best element to locally swap.

Recall that CP has a complexity evaluated to $O(\max\{m\theta, n\})$. CCP uses the constructive procedure CP to produce an initial feasible solution (Step 1). In the main step (Step 2), LocalSwapSearch procedure, takes $2m$ operations to update the used resources, one operation to compute the value $O(S)$. One takes $\theta(2m+1)$ operations to select a class and $n \times (2m+1) \times \theta$ operations for all the classes. So, the total operations of CCP in addition to the complexity of CP is evaluated to $MaxIter \times O(\theta(2m+1) \times n + (m+n) + \theta(2m+1))$ which is equivalent to $O(\theta(m+n))$. Hence, the worst case complexity of CCP is equivalent to $O(\theta \max\{m, n\})$.

A derived algorithm using penalties and normal transformations

In this section, we describe the main principle of the derived approach using (i) a penalty stage strategy and (ii) a normalized stage one. The algorithm starts by a feasible solution, namely S^* (obtained by applying the constructive procedure CP). The last solution is considered as the best feasible solution, obtained up to now, without using any penalty.

The derived algorithm (denoted Der_Algo) can be viewed as a two-phase procedure: it uses (i) a penalized phase and (ii) a normalized configuration one.

- On the one hand, the penalized phase is applied if the current solution cannot be improved after a certain number of iterations. In this case, a penalty parameter is used in order to transform the profits of the objective function. Starting with the new configuration (which remains a feasible one for the original problem), the process consists in finding a good neighbourhood for improving locally the current configuration.
- On the other hand, the normalized phase is used in order to transform the last penalized configuration into a normal feasible solution. The obtained solution is normalized because the profits of its objective function are set equal to the original profits (corresponding to the original problem instance).

Description of the derived algorithm: Der_Algo

The main steps of the derived algorithm are described in Box 3. The algorithm starts by applying CP to obtain an initial feasible solution. The configuration of the current solution is

Box 3 A derived algorithm using the penalty and the normal strategies: Der_Algo

```
Input: A solution S with value O(S), π, Δ and D.
Output: A best solution S* with value O(S*).
-----------------------------------------------------------
Initialization.
  Set S* ← S:= CP() and V(ρ) ← O(S*);
  Set Δ = D ← 0; /* Initializing the depth and the diameter
  parameters */
  Set phase ← Normal_Phase;/*Initializing the phase
  parameter */

Main step.
  While not (StoppingCondition()) Do
    S:= CCP(S)/* Using a LocalSwapSearch to improve
    the initial solution */
    If V(ρ) ⩽ O(S) then
      If (phase = Normal_Phase) then /* Stage without
      using any penalty factor */
        Set S* ← S, V(ρ) ← O(S*);
      Else
        Set S:= Normalize(S, ρ, π)
        /*Put back the solution configuration to its
        ordinary form */
        Set S* ← S, V(ρ) ← O(S*);
      EndIf
    Else
      If (phase = Normal_Phase) then
        S ← Penalize(S, V(ρ), π, Δ);
        /*Applying the penalization strategy to the current
        solution */
      Else /*If the current stage is a Penalize_Phase */
        Set S ← Normalize(S, ρ, π), V(ρ) ← O(S);
        S ← Penalize(S, V(ρ), π, Δ);
      EndIf
    EndIf
    Increment(D);
    Set Δ ← Get_Depth(Δ, D, n); /*A random fixation of
    depth */
  EndWhile

  Return S* with value O(S*);
```

stored in the vector ρ. The best solution (value) $S^*(O(S^*))$ is initially set equal to the initial solution (value) $S(O(S))$. The main loop (of the Main step) applies a Normal_Phase, which is the phase for which no penalty factor was introduced. The later phase performs a *local swapping search* in order to enhance the obtained solution. At each iteration of the main step, the best current solution is updated if the solution is improved. In this case, we can distinguish two cases:

- On the one hand, the stage is set to Normal_Phase and the solution is represented by S^* with value $O(S^*)$.
- On the other, if the stage is set to Penalize_Phase, then the Normalize () procedure is introduced in order to retrieve the solution's structure of the original problem, corresponding to the penalized solution.

This process is iterated for a certain number of iterations. By this way, we also can distinguish two cases: (i) the current solution was improved and (ii) the process is not able to

reach an improved solution. When the first case (i) is realized and the stage is running Normal_Phase, then the penalizing strategy—the Penalize () procedure—is called in order to modify the profits of the objective function. Furthermore, if condition (ii) is realized, then (a) the Normalize () procedure is applied in order to transform the penalized solution into a normal one and, (b) the Penalize () procedure is used on the later solution. Of course, the aim of the last step is to attempt to change the trajectory of the search process in order to explore other nonvisited space search.

In the following, we detail the two main procedures used in the Der_Algo algorithm, that is, the Penalize () procedure and the Normalize () one.

The Penalize **procedure**: It uses some parameters which are denned as follows:

- $0<\pi<1$: penalty coefficient,
- ρ: current solution vector in the penalized phase,
- Δ: the depth parameter for penalization,
- D: diameter parameter for exploration.

The procedure operates as follows:

Initialization

- The initial solution to penalize S is set equal to S^* and the current structure denoted ρ (which can be modified) is initially set to S;
- Let *Counter* (equal to zero) be the variable used in order to control the depth parameter Δ(i.e. $0 \leqslant Counter \leqslant \Delta$);

The main loop. it starts depth exploration to penalize

- While ($Counter \leqslant \Delta$) do
  - $i \leftarrow GetClass()$; /* Random selection of a class */
  - $j_i \leftarrow \rho[i]$; /* Index of an element in J_i^* /
  - $v_i\rho[i] \leftarrow \pi \times v_{i\rho[i]}$;
  - $O(S) \leftarrow O(S) - v_i\rho[i] + \pi \times v_{i\rho[i]}$;
  - Increment(Counter);
- Return S with value $O(S)$ as the penalized current solution.

The Normalize () **procedure**: We recall that the Normalize () procedure is applied in order to normalize any improved penalized solution produced by Penalize () procedure. The procedure works as follows:

Initially, the procedure uses some parameters which are defined as follows:

- $0<\pi<1$: penalty coefficient,
- ρ: current solution vector in the penalized phase and $V(\rho)$ it's value

The procedure operates as follows:

Initialization

- The initial solution to normalize S^* is setting equal to S and the solution value $V(\rho)$ is set equal to $O(S^*)$;

The main loop

- For $i = 1, \ldots, n$ do
  - $j_i \leftarrow \rho[i]$; /* to normalize the current component of the i-th class */
  - $O(S^*) \leftarrow O(S^*) - v_{ij} + (1/\pi) \times v_{i\rho[i]}$; /* to update the profits */
  - $v_{i\rho[i]} \leftarrow (1/\pi) \times v_{i\rho[i]}$;

We can remark that the derived algorithm uses other parameters. The first parameter is called *depth parameter* (denoted Δ) which permits to fix a number of items to penalize. The second parameter is called *diameter parameter* (denoted D) which is introduced in order to control the space search of some better obtained solutions up to now. For instance, the later space search represents some different configurations having the same solution value. Third and last, the parameters called *penalty coefficients* (denoted π) that are applied to the objective function. In this case, if the obtained configuration is feasible for the penalized problem, then it necessarily represents a feasible solution for the original problem.

The complexity of this algorithm is presented as follows. First, (Bloc 1), Der_Algo starts by calling CP to construct a feasible initial solution. We know that its complexity is of $O(\max\{m\theta, n\})$. Second (Bloc 2), we apply CCP to improve a current solution obtained by applying CP and its complexity has been evaluated to $O(\theta\max\{m, n\})$. Normalize () procedure takes at worst θ operations to put back a penalized solution to its normal configuration and has a complexity of $O(\theta)$. Penalize () procedure takes, as well, at worst θ operations to use a penalty factor for a current solution with a complexity of $O(\theta)$. So, the total operations taken by Der_Algo is bounded by $MaxIter \times ((\theta m + n) + (\theta(m+n) \times \theta)$. Hence the worst-case complexity of Der_Algo is equivalent to $O(\theta^2\max\{n, m\})$.

Computational results

The purpose of this section is two-fold: (i) to evaluate the performance of the CP and CCP and (ii) to determine a good trade-off between the running time and the used parameters for the derived algorithm (Der_Algo): the maximum number of iterations, the depth parameters, the diameter parameter' and the penalties ones.

This section is organized as follows. First, we evaluate the performance of both CP and CCP. For a set of problems extracted from the literature, we compare the results obtained by both algorithms to the optimal solution (or the best solution found up to now), and to the results of Moser *et al*[20] and Khan *et al*[21] approaches. Second, we present the performance of Der-Algo and reveal the importance of the used parameters. In the same section, we indicate the degree of improvement provided by Der_Algo over other approaches.

In our computational results, CP, CCP and Der_Algo are coded in C++, and run on a Ultra-Sparc10 (250 MHz and with 128 MB of RAM).

Performance of CP and CCP

To evaluate the performance of CP and CCP, we use the test problems of Khan *et al.*[21] (We have made these instances publicly available from ftp://panoramix.univ-paris1.fr/pub/CERMSEM/hifi/OR-Benchmark.html, hoping to aid further development of exact and approximate algorithms for the MMKP). These problems span a variety of instances varying from small- tolarge-sized instances. The optimal solution value for some of these instances, referred to as I01, ..., I06 in Table 1, is known. For the other instances, referred to as I07, ..., I13 in Table 1, we report the best solution value published by Khan *et al.*[21] For each instance, we report the number n of classes, the number r_i of items of each class, and N the total number of items of each instance representing $\sum_{i=1}^{n} r_i$.

Table 1 Test problem details

Inst.	n	r_i, $i=1, \ldots, n$	N
I01	5	5	25
I02	5	10	50
I03	10	15	150
I04	10	20	200
I05	10	25	250
I06	10	30	300
I07	10	100	1000
I08	10	150	1500
I09	10	200	2000
I10	10	250	2500
I11	10	300	3000
I12	10	350	3500
I13	10	400	4000

The results obtained by Moser *et al* and Khan *et al* are reported in Table 2. The results of Moser *et al* approach are provided in columns 5 and 6. The results of Khan *et al* algorithm are represented by columns 3 and 4. Column 2 contains the optimal-(or best-) solution value of the problem. Columns 4 and 6 show the percentage deviation of the solution value from the optimum (or best value) noted herein, *Dev*, and computed as follows: $Dev = (1-(A(I)/Opt(I)(or\ Best)) \times 100$, where $A(I)$ and $Opt(I)$ (resp. *Best*) denote the approximate (the solutions of column 3 or 5) and the optimal (resp. best) solutions of instance I.

The CP and the CCP solutions for I1, ..., I13 are reported in Table 3. The results of CP are provided in columns 3–5. Column 3 contains the solution value (denoted CP_{sol}). Column 4 shows the percentage deviation (denoted *Dev*) between the usage of the CP-yielded solution and the optimal (or best) solution, defected *Opt/Best*. Column 5 displays the CP run time (denoted T and measured in seconds). Columns 6–8 report the results of CCP. Column 6 provides the CCP solution (denoted CCP_{sol}), column 7 computes the corresponding deviation from the optimum (or the best solution) and column 8 displays the CCP run time.

This section can be considered as a preliminary experiment in which we compare the results of CP and CCP. Before comparing the results of both algorithms, let us analyse the behaviour of both solution approaches of the literature. From Table 2 we can observe that Khan *et al* algorithm (denoted *KLMA*) outperforms Moser's approach (denoted *Moser*: the reported solutions are taken from Khan *et al*[21]). In this case, *KLMA* produces a percentage deviation varying in the interval [0, 4.45%] and with an average percentage of 1.46%.

Now we return to analyse CP and CCP. Summarized results of CP and CCP appears in Table 3. We can observe that CCP produces better solutions than over all problems, CP at the expense of a slightly larger computational time.

Table 2 Perfomance of Khan *et al*[21] and Moser *et al*[20] algorithms on all problems

Inast.	*Opt/Best*	$KLMA_{\text{sol}}$	*Dev*	$Moser_{\text{sol}}$	*Dev*
I01	173.00	167.00	3.47	151.00	12.72
I02	364.00	354.00	2.75	291.00	20.05
I03	1602.00	1533.00	4.31	1464.00	8.61
I04	3597.00	3437.00	4.45	3375.00	6.17
I05	3949.59	3899.10	1.28	3905.70	1.11
I06	4799.30	4799.30	0.00	4115.20	14.25
I07	23 983.00*	23 912.00	1.02	23 556.00	2.50
I08	36 007.00*	35 979.00	0.11	35 373.00	1.79
I09	48 048.00*	47 901.00	0.31	47 205.00	1.75
I10	60 176.00*	59 811.00	0.68	58 648.00	2.61
I11	72 003.00*	71 760.00	0.45	70 532.00	2.16
I12	84 160.00*	84 141.00	0.03	82 377.00	2.13
I13	96 103.00*	96 003.00	0.10	94 166.00	2.02
Average			1.46		5.99

The symbol * means that the optimal solution is not known.

Table 3 Performance of both CP and CCP algorithms on all problems

Inst.	*Opt/Best*	CP_{sol}	*Dev*	*T*	CCP_{sol}	*Dev*	*T*
I01	173.00	161	6.94	<0.01	161.00	6.94	<0.01
I02	364.00	284.00	21.98	<0.01	341.00	6.32	<0.01
I03	1602.00	1414.00	11.74	<0.01	1511.00	5.68	<0.01
I04	3597.00	3135.00	12.84	<0.01	3397.00	5.56	<0.01
I05	3949.59	3065.40	22.40	<0.01	3591.59	9.06	0.03
I06	4799.30	3749.89	21.87	0.01	4567.90	4.82	0.02
I07	23 983.00*	19 667.00	18.59	0.02	23 753.00	1.68	0.16
I08	36 007.00*	28 461.00	20.98	0.05	35 485.00	1.48	0.40
I09	48 048.00*	38 389.00	20.10	0.06	47 685.00	0.76	0.65
I10	60 176.00*	48 361.00	19.69	0.05	59 492.00	1.21	1.11
I11	72 003.00*	58 008.00	19.53	0.08	71 378.00	0.98	1.35
I12	84 160.00*	68 027.00	19.18	0.09	83 293.00	1.04	1.70
I13	96 103.00*	78 309.00	18.52	0.09	95 141.00	1.00	2.15
Average			18.03	0.03		3.58	0.58

The symbol * means that the optimal solution is not known. The symbol < means that the computational time is neglected.

CCP produces, in less than 0.58 s, reasonable quality results. It is on average 3.06% of the optimum (or best-obtained solution). It occasionally yields poor results with a worst-case of 8.93%. Thus, it is a useful starting point for more complex procedures.

Performance of Der_Algo

Generally, when using approximate algorithms to solve optimization problems, it is well known that different parameter settings for the method lead to results of variable quality. Herein, Der_Algo involves four decisions: the way of choosing the depth parameter Δ, the number of iterations, *Max_Iter*, to carry out, the way of controlling the space search represented by the diameter parameter D, and the values attributed to the penalty parameter π (in our case, the penalty parameter is the same for all profits). In what follows, a different adjustment of the method's parameters would lead to a high percentage of good solutions. But this better adjustment 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 run time.

First, in order to find the right value of Δ we have explored three strategies:

- A bigger value was assigned to Δ, that is, by fixing Δ in the interval $[6, \ldots, 10]$;
- An intermediate value was assigned to Δ, that is, by setting Δ to 5;
- A smaller value was assigned to Δ, that is, by varying Δ in the interval $[1, \ldots, 4]$.

Limited computational results showed that the variation of Δ in the interval $[1, \ldots, 5]$ produced a good improvement of the solution quality. For the complementary interval $[6, \ldots, 10]$, the algorithm was not able to produce a better solution, but it consumed more computational time. Finally, the best results were obtained for the second case and this value of $\Delta = 5$ was retained and used in what follows.

Table 4 The behaviour of Der_Algo when varying the number of iterations *Max_Iter*

Max_Iter	*2*	*5*	*8*	*10*
Av.Dev	1.81	0.92	0.68	0.61
Av.T	1.90	1.90	4.10	6.50

Second, in order to find a good compromise between the quality of the solutions and the computational time, we have introduced a variation for the maximum number of iterations *Max_Iter*. In this case, we have tested *Max_Iter* with values taken from the discrete interval {2, 5, 8, 10}. Limited computational results revealed that a bigger value of *Max_Iter* does not necessarilly generate a better solution, but the computational time increases.

Table 4 shows the quality of the results obtained when Der_Algo is applied with the following parameters: $\Delta = 5$, $D = 5$ and $\pi = 0.70$ (below, we shall discuss the choice of the values associated to D and π). Using these later values, as shown in Table 4, we can observe that the quality of the results (denoted *Av. Dev*: Line 2) varies between 0.61 and 1.81%. The better average deviation is obtained when fixing *Max_Iter* to 10 with a largest average computational time (denoted *Av. T.* : Line 3).

Third, by fixing the values of the parameters Δ and *Max*, we now try to fix the value of the diameter parameter D in order to control and limit the space search. Indeed, the later parameter permits us to consider a certain diversification of the solutions when several best solutions (for instance, these solutions have the same value but with different configurations) are reached by the algorithm. Table 5 reports the

Table 5 The behaviour of Der_Algo when varying the value of the parameter D

D	3	5	7	10
Av.Dev	1.14	0.61	0.63	0.70
Av.T	2.50	6.50	7.40	9.50

Table 6 The behaviour of Der_Algo when varying the value of the parameter π

π	0.50	0.70	0.80	0.90
Av.Dev	0.70	0.61	0.64	0.70
Av.T	6.40	6.50	6.90	9.50

quality of the obtained results when D is varied in the discrete interval {3, 5, 7, 10}.

We can remark that the average deviations vary between 0.61 and 1.14%, and the better result is obtained for $D=5$. The same table shows that if the value of D is very small or very large, then the used diversification is less or more important. We think that for the small values of D, the generated space is not sufficient for exploring good solutions. For the largest value, we think also that the algorithm explores a very large space and so, the guided search is not able to locate a good direction in order to improve some visited solutions. From Table 5, we can conclude that an intermediate value for D maintains the high quality of the solutions.

Fourth and finally, we analyse the behaviour of Der_Algo when varying the parameter π. Table 6 summarizes the results obtained by Der_Algo. From the later table, we observe that Der_Algo gives good-quality results for the value 0.70. It yields an average deviation of 0.61%. Note that, for the other values, the algorithm degrades the solution quality. In addition, Der_Algo is very fast for the later value. Its average run time is equal to 6.5 s and it gives better solutions within small computational times (compared to the results of both values 0.8 and 0.9). We can conclude that it is not necessary to use the smallest or the largest value of π for producing good solutions.

In what follows, we give the solution values produced by Der_Algo and we compare its performance to that of Khan *et al*s approach, referred to herein as KLMA (see Table 2). Specifically, we consider the version of the algorithm for which the parameters are fixed as follows: $\Delta=5$, $D=5$, $\pi=0.7$, and *Max_Iter* is equal to 10.

The performance of Der_Algo is assessed using the problem instances of Table 1. The results of the algorithm are displayed in Table 7. For each instance, we report the solution value (denoted Der_Algo_{sol}), the deviation (denoted Dev) between the obtained solution and the solution that KLMA produced (in this case, the negative deviation $-\gamma$ means that the algorithm has an improvement of γ%), the run time (denoted T and measured in seconds), and the average deviation (resp. run time) it takes Der_Algo to reach the final solution (the last line of Table 7).

From Table 7, we observe that Der_Algo produces better solution values compared to those of KLMA. On average, it realizes an improvement of 0.68% from the solutions produced by KLMA. Indeed, the observed percentage improvement varies in the interval [0, 3.59%] for the treated instances. In addition, we can remark that the solutions are obtained under 1 min (especially for large-scale instances). Note that for the small instances, I01, ..., I06, Der_Algo improves significantly the solutions produced by CCP and it gives better results compared to the results of KLMA. For the other instances, I07, ..., I13 which are considered as large-scale problems, Der_Algo gives better solutions compared to the results produced by KLMA.

Table 7 Performance of Der_Algo compared to the results of KLMA algorithm, on the problem instances of Table 1

Inst.	$KLMA_{sol}$	Der_Algo_{sol}	*Dev*	*T*
I01	167.00	173°	−3.59	0.04
I02	354.00	356.00▷	−0.56	0.04
I03	1533.00	1553.00▷	0.00	0.08
I04	3437.00	3502.00▷	−1.89	0.09
I05	3899.10	3943.22▷	−1.13	0.15
I06	4799.30	4799.30	0.00	0.21
I07	23 912.00*	23 983.00°	−0.30	1.50
I08	35 979.00*	36 007.00°	−0.08	2.17
I09	47 901.00*	48 048.00°	−0.31	5.50
I10	59 811.00*	60 176.00°	−0.61	7.47
I11	71 760.00*	72 003.00°	−0.34	13.35
I12	84 141.00*	84 160.00°	−0.02	22.41
I13	96 003.00*	96 103.00°	−0.10	31.64
Average			−0.68	6.50

The symbol ○ means that the optimal (or the best) solution value was attained and the symbol ▷ means that Der_Algo improves the solution produced by KLMA.

Conclusion

In this paper, we have proposed several approximate algorithms for solving the MMKP. The first algorithm is a constructive procedure applied for obtaining an initial solution for the problem. The second algorithm is an improved version of the constructive procedure, introduced for improving the quality of the solution. The third algorithm is based mainly upon a guided local search which uses a penalization strategy. The principle of the approach is to construct an initial solution and to tailor on it a neighbourhood search. The algorithm can be viewed as a two-stage procedure: (i) the first stage is applied in order to penalize a current solution and, (ii) the second stage is used

in order to normalize and to improve the quality of the solution given by the first-stage. Computational results show that the algorithm generates high-quality solutions within small computing times.

Acknowledgements—Many thanks to anonymous referees for their helpful comments and suggestions contributing to improving the presentation and the contents of the paper.

References

1 Chen G, Khan S, Li KF and Manning E (1999). Building an adaptive multimedia system using the utility model. *Parallel and Distributed Processing*. Lecture Notes in Computer Sciences, Vol. 1586. Springer: Berlin, pp 289–298.
2 Watson RK (2001). *Packet Networks and optimal admission and upgrade of service level agreements: applying the utility model.* MSc thesis, Department of ECE, University of Victoria, Canada.
3 Voudouris C and Tsang EPK (1996). Partial constraint satisfaction problems and guided local search for combinatorial. In: *Proceedings of Practical Application of Constraint Technology (PACT'96)*, pp 337–356. http://cswww.essex.ac.uk/csp/gls-papers.html
4 Voudouris C and Tsang EPK (1999). Guided local search and its application to the travelling salesman problem. *Eur J Opl Res* **113**: 469–499.
5 Martello S and Toth P (1990). *Knapsack Problems: Algorithms and Computer Implementations*. Wiley: Chichester, England.
6 Chu P and Beasley JE (1998). A genetic algorithm for the multidimensional knapsack Problem. *J Heuristics* **4**: 63–86.
7 Martello S and Toth P (1987). Algorithms for knapsack problems. *Ann Discrete Math* **31**: 70–79.
8 Balas E and Zemel E (1980). An algorithm for large zero-one knapsack problem. *Opns Res* **28**: 1130–1154.
9 Fayard D and Plateau G (1982). An algorithm for the solution of the 0-1 knapsack problem. *Computing* **28**: 269–287.
10 Martello S and Toth P (1988). A new algorithm for the 0-1 knapsack problem. *Mngt Sci* **34**: 633–644.
11 Pisinger D (1997). A minimal algorithm for the 0-1 knapsack problem. *Opns Res* **45**: 758–767.
12 Martello S, Pisinger D and Toth P (1999). Dynamic programming and strong bounds for the 0-1 knapsack problem. *Mngt Sci* **45**: 414–424.
13 Freville A and Plateau G (1994). An efficient preprocessing procedure for the multidimensional 0-1 knapsack problem. *Discrete Appl Math* **49**: 189–212.
14 Shih W (1979). A branch and bound method for the multi-constraint zero-one knap-sack problem. *J Opl Res Soc* **30**: 369–378.
15 Nauss MR (1978). The 0-1 knapsack problem with multiple-choice constraints. *Eur J Opl Res* **2**: 125–131.
16 Khan S (1998). *Quality adaptation in a multi-session adaptive multimedia system: model, algorithms and architecture*. PhD Thesis, Department of Electronical and Computer Engineering, University of Victoria, Canada.
17 Pisinger D (1999). An exact algorithm for large multiple knapsack problems. *Eur J Opl Res* **114**: 528–541.
18 Toyoda Y (1975). A simplified algorithm for obtaining approximate solution to zero-one programming problems. *Mngt Sci* **21**: 1417–1427.
19 Dantzig GB (1957). Discrete variable extremum problems. *Opns Res* **5**: 266–277.
20 Moser M, Jokanović DP and Shiratori N (1997). An algorithm for the multidimesional multiple-choice knapsack problem. *IEICE Trans Fundamentals Electron* **80**: 582–589.
21 Khan S, Li KF, Manning EG and Akbar MDM (2002). Solving the knapsack problem for adaptive multimedia systems. *Stud Inform* **2**: 154–174.
22 Hansen P (1986). The steepest ascent mildest descent heuristic for combinatorial programming, Presented at the Congress on Numerical Methods in Combinatorial Optimization, Capri, Italy.
23 Glover F (1986). Future paths for integer programming and links to artificial intelligence. *Comput Opns Res* **13**: 533–549.
24 Glover F and Laguna M (1997). *Tabu search*. Kluwer Academic Publishers: Boston, USA.
25 Voudouris C and Tsang EPK (1995). *Guided local search*. Technical Report CMS-247, Department of Computer Science, University of Essex, England.
26 Tsang EPK and Voudouris C (1997). Fast local search and guided local search and their application to British Telecom's worforce scheduling problem. *Opns Res Lett* **20**: 119–127.
27 Kilby P, Prosser P and Shaw P (1997). Guided local search for the vehicle routing problem with time windows. In: Voss S, Martello S, Osman IH and Roucairol C (eds). *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*. Kluwer Academic Publishers: Dordrecht, pp 473–486.
28 Faroe O, Pisinger D and Zachariasen M (2003). Guided local search for the three-dimensional bin packing problem. *INFORMS J Comput* **15**: 267–283.

Received March 2003;
accepted April 2004 after 2 revisions