Computers & Industrial Engineering 99 (2016) 280–286

Contents lists available at ScienceDirect

Computers & Industrial Engineering

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

A tree search based combination heuristic for the knapsack problem with setup

Mahdi Khemakhem [a,*], Khalil Chebil [b]

[a] College of Computer Engineering and Science, Prince Sattam Bin Abdulaziz University, Kingdom of Saudi Arabia
[b] LOGIQ, University of Sfax, Tunisia

ARTICLE INFO

Article history:
Received 19 July 2015
Received in revised form 29 March 2016
Accepted 26 July 2016
Available online 27 July 2016

Keywords:
Knapsack problems
Setup
Tree search
Combination
Filter-and-fan metaheuristic
Avoid duplication

ABSTRACT

Knapsack Problems with Setups (KPS) have received increasing attention in recent research for their potential use in the modeling of various concrete industrial and financial problems, such as order acceptance and production scheduling. The KPS problem consists in selecting appropriate items, from a set of disjoint families of items, to enter a knapsack while maximizing its value. An individual item can be selected only if a setup is incurred for the family to which it belongs. In this paper, we propose a tree search heuristic to the KPS that generates compound moves by a strategically truncated form of tree search. We adopt a new avoid duplication technique that consists in converting a KPS solution to an integer index. The efficiency of the proposed method is evaluated by computational experiments involving a set of randomly generated instances. The results demonstrate the impact of the avoiding duplication technique in terms of enhancing solution quality and computation time. The efficiency of the proposed method was confirmed by its ability to produce optimal and near optimal solutions in a short computation time.

1. Introduction

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

Our interest in this model was originally motivated by practical problems at a production project with a leading manufacturer and supplier of agro-alimentary glass packing industry. This company produces several types of products, including bottles, flacons, and pots. The most important phase in the manufacturing process, is the phase of shaping. In fact, to change the production from one product family to another, the production machinery must be set up and molds must be changed in the molding machine. These changes in the manufacturing process require significant setup time and costs. Assume at time T, the company receive some orders (jobs), wich belong to N product families. Each product family i, has n_i jobs. Also assume that these jobs should be produced in the next planning period and the company's manufacturing capacity is fixed and can't be changed in the short term. Accordingly, the company needs to decide on how to choose orders so as to maximize the total profit. This represents a typical case involving a knapsack problem with setup model that can be used to solve this problem.

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

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

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

$$s.t. \quad \sum_{i=1}^{N}\sum_{j=1}^{n_i} a_{ij}x_{ij} + \sum_{i=1}^{N} d_i y_i \leqslant b \tag{2}$$

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

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

* Corresponding author.
E-mail addresses: m.khemakhem@psau.edu.sa (M. Khemakhem), khalil.chebil@issatm.rnu.tn (K. Chebil).

http://dx.doi.org/10.1016/j.cie.2016.07.021
0360-8352/

Eq. (1) represents the KPS objective function. Constraint (2) ensures that the weight of selected items in the knapsack, including their setup capacity consumption, does not exceed knapsack capacity b. Constraints (3) guarantee that each item is selected only if it belongs to a class that has been set up. Constraints (4) require the decision variables to be binary, where x_{ij} refers to the item variables and y_i to the setup variables. In fact, y_i is equal to 1 if the knapsack is set up to accept items belonging to class i and is equal to 0 otherwise. x_{ij} is equal to 1 if the item j of the class i is placed in the knapsack and is equal to 0 otherwise.

The KPS is a generalization of the standard 0–1 Knapsack Problem (KP), which is known to be an NP-hard problem (Kong, Gao, Ouyang, & Li, 2015; Martello & Toth, 1990). In fact, it is not difficult to verify that a special case of the KPS, where $N = 1$, is equivalent to the KP. The works of Martello and Toth (1990) and Kellerer, Pferschy, and Pisinger (2004) provide a thorough overview of the research so far performed on the KP and its variations which reflect its ability to closely represent and respond to real world problems. Chajakis and Guignard (1994) consider a similar problem of KPS, where the setup cost f_i and profit c_{ij} of an item j of a class i can be negative or non-negative. An extra constraint is added to make sure that if the knapsack is set up for a class i, at least one item of this class must be selected. Chajakis and Guignard (1994) propose a dynamic programming algorithm and two versions of a two-phase enumerative scheme. The experiments show that the dynamic programming approach is most efficient for correlated instances with small knapsack capacity. Akinc (2004) presents a set of algorithms based on several properties of the linear programming relaxation of a special case of KPS with no setup capacity consumption, called the Fixed Charge Knapsack Problem (FCKP). He showed that if the FCKP is solved as an LP and if all the y_i variables obtained are integers, then the optimal solution is obtained by solving the KP to optimality allocate the remainder capacity to all x_{ij} for which $y_i = 1$. Yang (2006) has, however, demonstrated the inadequacy of the last proposal by presenting a counter-example. Mclay and Jacobson (2007) provide three dynamic programming algorithms to solve the Bounded Setup Knapsack Problem (BSKP) in a pseudo-polynomial time. The latter were not, however, practical for solving large problem instances. Yang (2006) developed an effective branch-and-bound algorithm and proposed a heuristic method for the KPS. A thorough survey of the literature on the KSP has recently been presented in the work of Michel, Perrot, and Vanderbeck (2009) who provided an extension of the branch-and-bound algorithm proposed by Horowitz and Sahni (1974).

In this paper, we present a tree search based combination (TSC) heuristic to solve the KPS. The results from experimental assays on a randomly generated set of benchmark problems demonstrate the effectiveness of our approach. The rest of this paper is organized as follows: Section 2 presents the proposed algorithm. Section 3 evaluates the efficiency of the TSC algorithm using a set of randomly generated instances and compares its performance to CPLEX 12.5. Section 4 concludes by providing a summary and perspectives for future research.

2. Tree search based combination for the KPS

In this section, we start by presenting some preliminary considerations in the knapsack problem with setup that are relevant to the design of the proposed algorithm. We then move to describe the general TSC procedure.

2.1. Preliminary considerations

As mentioned earlier, the knapsack problem is a special case of the KPS (where $n = 1$). The special structure of KPS allows us to fix the setup variables ($y_i = 0$ or $y_i = 1$) needed to transform the KPS into a KP. Let's consider a set of item classes $Y = \{i \in \{1,\dots,N\}/y_i = 1\}$, where the knapsack is set up to accept items belonging to each class in Y. We define the $KPS[Y]$ problem as a sub-problem of KPS. $KPS[Y]$ is a knapsack problem with a capacity $b - \sigma$ and an objective function minimized by a negative integer setup cost δ. Then, the $KPS[Y]$ can be formulated by a KP 0–1 linear program as follows:

$$\text{Max } z = \sum_{i\in Y}\sum_{j=1}^{n_i} c_{ij}x_{ij} + \delta \tag{5}$$

$$s.t. \quad \sum_{i\in Y}\sum_{j=1}^{n_i} a_{ij}x_{ij} \leqslant b - \sigma \tag{6}$$

$$x_{ij} \in \{0,1\} \qquad \forall i \in Y, \quad \forall j \in \{1,\dots,n_i\} \tag{7}$$

where

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

With these considerations in mind, we only fix the setup variables y_i to 1 or to 0 and employ CPLEX to determine the best values for x_{ij}, which yields a feasible solution to the KPS. Thus, our neighborhood strategy focuses on finding the optimal combination of the setup variables Y^*.

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

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

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

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

and

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

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

2.2. The TSC approach

The TSC approach is closely similar to the Filter-and-Fan (F&F) method which was initially proposed in Glover (1998) as a method for refining solutions obtained by scatter search. The F&F approach consists in the integration of the filtration and sequential disper-

sion of candidate list strategies used in the tabu search metaheuristic (Glover & Laguna, 1999). The method combines a local search strategy that identifies a local optimum with a F&F procedure that explores larger neighborhoods to overcome local optimality. Once a new best solution is found in the F&F process, the method switches to the local search phase to identify a local optimum that will be taken as a new root node to the F&F tree for another run of the F&F approach.

The F&F method was further extended in Rego and Glover (2002) as an alternative to ejection chain procedures and as a means for creating combined neighborhood search strategies. This method has been successfully used to solve several problems, such as the facility location (Greistorfer & Rego, 2006), protein folding (Rego, Li, & Glover, 2011), job shop scheduling (Rego & Duarte, 2009), and capacitated minimum spanning tree (Rego & Mathew, 2011) problems. This method has also been recently (Khemakhem, Haddar, Chebil, & Hanafi, 2012) reported to be highly effective F&F for the 0–1 multidimensional knapsack problem.

Like the F&F model, the TSC model can be illustrated by means of a neighborhood tree where branches represent sub-moves and nodes identify the solutions produced by these moves. Fig. 1 represents an illustration of the general process of the TSC algorithm. The TSC method operates by progressively extending the tree level by level while keeping only a subset of potentially η_1 good nodes (black nodes) as candidates for further exploration. The TSC algorithm starts with an empty solution S_0, and then proceeds by choosing the best η_1 neighbor solutions to S_0 to be members of the first level of the tree. Let a level index k and $L(k)$ be the set of η_1 best solutions at level k. To create the next level $k+1$ of the tree, the TSC method selects a subset of η_2 best neighbor solutions for each solution of $L(k)$ to generate $\eta = \eta_1 \times \eta_2$ trial solutions for level $k+1$ (as a result of applying η_2 moves to each of the η_1 solutions at level k). Fig. 1 shows that at level 2 only nodes S_4, S_7 and S_9 were selected as candidates to create level 3. In fact, these nodes represent the η_1 best solutions at level 2. The method stops branching as soon as the maximum number of level L_{max} is reached.

Algorithm 1 describes the TSC procedure. It starts by computing the maximum number of the tree levels L_{max}, which is determined by solving the linear relaxation of the KPS model noted LKPS. In fact, after several test runs with different values of L_{max}, we decide in favor to use the number of non-zero setup variables in the optimal solution of the LKPS model as L_{max}. The linear relaxation of the KPS model is obtained by replacing constraint (4) with $0 \leqslant x_{ij}$ and $0 \leqslant y_i \leq 1$.

As for the usual heuristics and metaheuristics, the TSC algorithm requires the definition of a neighborhood structure using simple moves so as to produce a set of neighbor solutions and explore the search space. In our implementation, we use an Add move which consists in commuting the value of one setup variable from 0 to 1. Thus, all solutions at a level k have exactly selected k setup variables. To describe the TSC algorithm, we denote s_k^i the ith solution at level k and $V(s,\eta)$ the set of the η best neighbor solutions resulting from the application of the η best add moves to solution s.

The TSC approach is equipped with a tabu list aiming to drive the search and keep the method from generating duplicated solutions in the same level. It can be viewed as a precondition for creating an appropriate level of diversification. We propose a new avoid duplication technique, which will be described in the coming section. The TSC algorithm finds a KPS solution in polynomial $O(N^2)$ time in the worst case ($L_{max} = N$) and in $O(N)$ time in the best case ($L_{max} = 1$).

Algorithm 1. The general TSC procedure

```
Data: η1, η2
Result: Solution S
Compute the maximum number of levels Lmax;
Generate the first level L(1) of the TSC tree;
S = the best solution in L(1);
for k ← 2 to Lmax do
    L'(k) = ∅;
    T=∅ //initialize the tabu list;
    for i ← 1 to η1 do
        //add the best η2 neighbor solutions of s^i_{k-1} to L'(k)
        L'(k) = L'(k) ∪ {V(s^i_{k-1}, η2) ∉ T };
        T = T ∪V(s^i_{k-1}, η2);
    end
    L(k) = the η1 best solutions in L'(k) ;
    if (the best solution in L(k) is better than S) then
        Update S;
    end
end
```

2.3. Avoid duplication

As previously mentioned, the TSC method progresses level by level, without backtracking; it only explores the most promising nodes at each level. Fig. 2 provides an example of a TSC tree used to perform a compound move from the root node to an enhanced solution. Gray nodes represent the η_1 selected solutions at each level; the black nodes represent duplicated solutions generated in one level (i.e. Level 3). To deal with this duplication problem, we define a vector T of visited solutions at each level of the tree. According to Fig. 2 and at the end of level 2 generation, we obtain a tabu list T = $\{(1|0|1|0|0), (1|1|0|0|0), (0|1|1|0|0), (0|0|1|1|0), (0|1|0|1|0), (1|0|0|1|0)\}$. The storage of binary solutions requires large memory space. In order to reduce the storage requirement, we adopt a new space reduction technique consisting of converting a solution to an integer index. In fact, given a solution $S = (0|1|0|1|1) = \{2,4,5\}$, S represents a solution of a KPS instance with $N = 5$ classes (where classes are indexed from 1 to 5). We can consider S as a $<$ 3-*combinations* $>$ of N classes. Combinations can refer to the combination of N things taken k at a time. The number of $<$ *k-combinations* $>$ of N classes is equal to the binomial coefficient $\binom{N}{k} = \frac{N!}{k!(N-k)!}$ (see Algorithm 2).

All $<$ *k-combinations* $>$ of N classes can be put in bijection with the natural numbers from 0 to $\binom{N}{k} - 1$. Algorithm 3 provides the index of a given solution S and, in turn, Algorithm 4 provides the combination that corresponds to an index (for a given N and k). The main purpose of this conversion is to provide a representation, each by a single number, of all $\binom{N}{k}$ possible $<$ *k-combinations* $>$ of N classes. In order to ensure that, we need to impose some order on the set of all $<$ *k-combinations* $>$ of N classes using a lexicographic ordering of the decreasing sequence of elements figuring in each combination. Table 1 represents an example of lexicographical ordering of all $<$ 3-*combinations* $>$ list of 5 classes. We

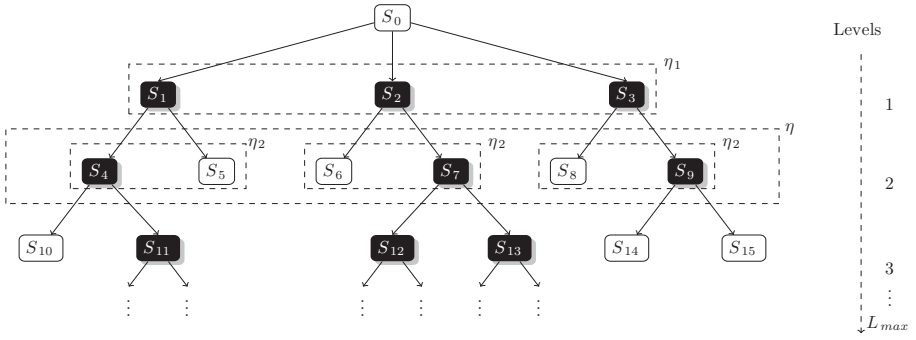

Fig. 1. Example of a TSC tree ($\eta_1 = 3, \eta_2 = 2$).

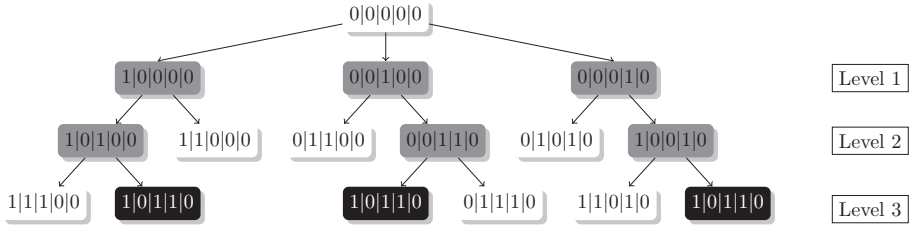

Fig. 2. Example of duplicate solutions in the TSC tree ($\eta_1 = 3, \eta_2 = 2$).

note that the order is lexicographical on the characteristic vector and not on the corresponding binary solution.

Using this technique allows us to significantly reduce the storage requirement. In fact, in the previous example, the tabu list T represents a list of $<$ 2-*combinations* $>$ of 5 classes generated at level 2. It can be represented using only the combination index of each solution, *T* = {8,9,7,4,5,6}.

We can also extend this technique to reduce the storage requirement of the TSC tree. Fig. 3 presents an equivalent representation of the TSC tree presented in Fig. 2 using the combination index.

Algorithm 2. Binomial

Data: N, k
Result: b
if $(N \leq 0 || k \leq 0)$ **then**
  $b \leftarrow 0$;
else
  $b \leftarrow 1$;
  for $i \leftarrow 0$ *to* $k-1$ **do**
    $b \leftarrow b \times (N - i)$;
    $b \leftarrow b \div (i + 1)$;
  end
end

Algorithm 3. SolutionToIndex

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

Algorithm 4. IndexToSolution

Data: $N, k, index$
Result: S
if $(index < 0 || N \leq 0 || k \leq 0)$ **then**
  break;
end
$b \leftarrow Binomial(N - 1, k - 1)$;
if $(index < b)$ **then**
  $IndexToSolution(N - 1, k - 1, index)$;
  ADD N to S;
else
  $IndexToSolution(N - 1, k, index - b)$;
end

Table 1
Lexicographical ordering of all < 3-*combinations* > list of 5 classes.

Index	Characteristic vector	Binary solution
0	{5,4,3}	[0,0,1,1,1]
1	{5,4,2}	[0,1,0,1,1]
2	{5,4,1}	[1,0,0,1,1]
3	{5,3,2}	[0,1,1,0,1]
4	{5,3,1}	[1,0,1,0,1]
5	{5,2,1}	[1,1,0,0,1]
6	{4,3,2}	[0,1,1,1,0]
7	{4,3,1}	[1,0,1,1,0]
8	{4,2,1}	[1,1,0,1,0]
9	{3,2,1}	[1,1,1,0,0]

3. Experimental results

The TSC algorithm was coded in *C language* and tested on a 2.1 *GHZ Intel Core*^{TM}*i*3. Due to the unavailability of benchmark instances in the literature, the performance of our algorithm was evaluated on a randomly generated list of benchmark instances that resembles the ones presented by Chajakis and Guignard (1994) and Yang (2006), with a total number of items $n_{tot} \in \{500, 1000, 2500, 5000, 10000\}$ and $N \in \{5, 10, 20, 30\}$ (available at https://goo.gl/Ge91cF). Setup cost and capacity consumption are given by:

$$f_i = -e_1\left(\sum_{j=1}^{n_i} c_{ij}\right) \tag{8}$$

$$d_i = -e_1\left(\sum_{j=1}^{n_i} a_{ij}\right) \tag{9}$$

where e_1 is uniform from [0.15, 0.25].

We choose a_{ij} uniformly from [10, 100] and $c_{ij} = a_{ij} + 10$ (a and c are strongly correlated). In order to make hard instances, we choose $b = 0.5 * \sum_{i=1}^{N}\sum_{j=1}^{n_i} a_{ij}$, the cardinality of each class n_i, for $i = 1, \ldots, N$, is in $[k - \frac{k}{10}, k + \frac{k}{10}]$ with $k = \frac{n_{tot}}{N}$.

After some preliminary test runs to balance the running time and quality of the results, we decided in favor of the following fixed set of TSC parameters: we selected $\eta_1 = \eta_2 = 5$ and fixed the CPLEX time limit to 2 s when solving each generated knapsack problem. Fig. 4 shows the results obtained for 20 subsets of test instances according with respect to the average percentage deviation from solutions obtained by TSC method with $\eta_1 = \eta_2 = 5$ (see Fig. 4 (a)) and the corresponding CPU deviation time in second (see Fig. 4 (b)). Each entry in the x axis represents a set of 10 problems

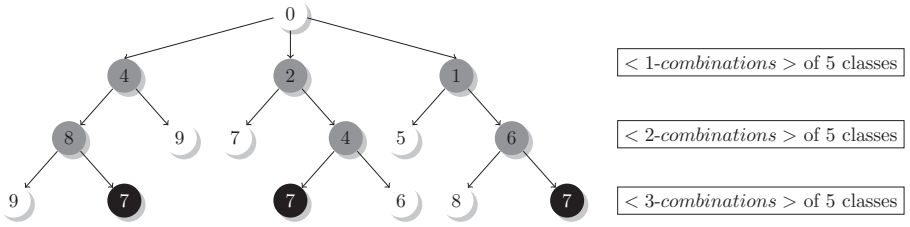

Fig. 3. Representing the TSC tree in Fig. 2 using combination index.

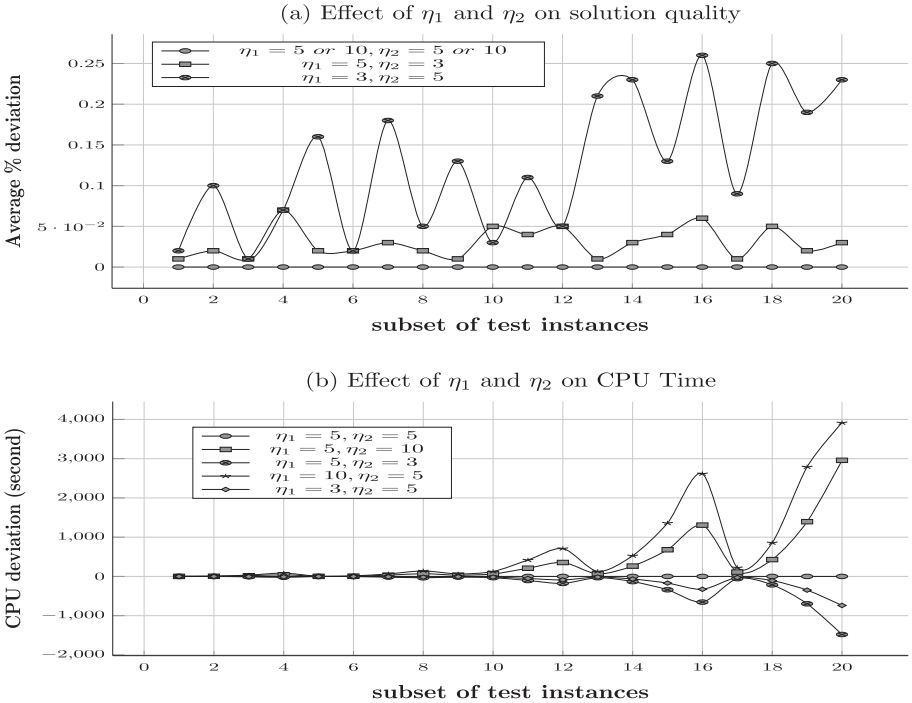

Fig. 4. Effect of TSC structural parameters.

Table 2
Performance of TSC algorithm compared to CPLEX 12.5

n_{tot}	N	$CPLEX_{obj}$	TSC_{obj}	$CPLEX_{UB}$	$CPLEX_{cpu}$ (s)	TSC_{cpu} (s)
500	5	110738	110735	110745	2225	5
	10	111446	111162	111446	1692	19
	20	139178	139173	139193	2037	84
	30	139524	139433	139531	3294	209
1000	5	199778	199773	199820	7789	12
	10	219433	219433	219485	5566	36
	20	226480	225293	226480	1876	160
	30	226536	225843	226654	8274	351
2500	5	555190	555190	555263	4490	161
	10	548500	547840	550306	16485	313
	20	501682	501716	504939	9840	1045
	30	555124	553184	556264	9041	1775
5000	5	1003022	1003006	1003085	14522	319
	10	1006494	1006294	1006592	8598	1320
	20	1006458	1004563	1009982	12743	3398
	30	1012004	1007560	1012251	4709	6539
10000	5	2231298	2231298	2231356	13335	556
	10	2012279	2012279	2013341	19615	2144
	20	2005891	2014718	2022815	27427	6974
	30	2015818	2003318	2016192	13829	14788

with a total number of items $n_{tot} \in \{500, 1000, 2500, 5000, 10000\}$ and $N \in \{5, 10, 20, 30\}$.

As expected, TSC method performance is consistently improved as η_1 and η_2 increases. Furthermore, by considering the computational effort, the pay off seems to be rather poor at the cost of the expense in computational time. To that end, a TSC parameters with $\eta_1 = \eta_2 = 5$ seems to provide a good compromise.

In Table 2, we report the results obtained by the TSC method as compared to the upper bound provided by CPLEX 12.5 when solving KPS. Each row summarizes 10 instances. The first two columns present the total number of items $n_{tot} = \sum_{N}^{i=1} n_i$ and the number of classes N. The next three columns provide the corresponding sum of values provided by the TSC algorithm (TSC_{obj}), the sum of values provided by CPLEX ($CPLEX_{obj}$), and the sum of the upper bounds provided by CPLEX ($CPLEX_{UB}$) which corresponds to the best known bound of all the remaining open nodes in the branch-and-cut tree. The next columns present the total running time of the TSC approach (TSC_{cpu}) and the total running time of CPLEX ($CPLEX_{cpu}$). Processing times (CPU's) are reported in seconds.

We note that CPLEX finds an optimal solution for 79 out of 200 problems. For the rest, CPLEX terminates with an error message (exceeds the capacity of RAM memory or exceeds the limit of CPU time). The values provided by CPLEX for those problems correspond to the values of the best feasible solutions found with a CPU time limit of 3600 s. Furthermore, the TSC algorithm provides a solution equal to CPLEX for 79 problems and provides a new best known value for 15 problems. The results presented in Table 2 show that the proposed method is highly effective for providing solutions that are on average at about 0.086%, 0.232%, 0.413%, 0.259% and 0.273% with respect to the upper bound provided by CPLEX for instances with 500, 1000, 2500, 5000 and 10000 variables. It is also noted that instances with larger N are more difficult to solve and that the running time of the TSC approach increases with the increase of N.

In order to properly evaluate the impact of the new technique of avoid duplication on the performance of our approach in terms of computation time and quality of solutions, the results obtained by the TSC method were compared to the ones achieved by another version of the TSC method with duplication TSC_{dup}; the results are presented in Figs. 5 and 6. Fig. 5 (resp. Fig. 6) shows five problem classes with $n_{tot} = \{500, 1000, 2500, 5000, 10000\}$, respectively, and for each problem class, the x axis represents the set of 40 problems while the y axis represents the sum of objective values (resp. the sum of CPU time) for the set of 40 problems. The results clearly demonstrate the impact of avoiding duplication on solution quality and computation time. Compared to the results obtained with the TSC method, the ones achieved with the TSC_{dup} method show that the objective value has decreased in 40 instances and increased in 8 instances and that the CPU time has increased in all instances. A complete and detailed version of the results is available at https://goo.gl/SjW1gy.

4. Conclusion

In this paper, we proposed a tree search based combination (TSC) heuristic for the knapsack problem with setup. The TSC is an iterative local search method that explores the solution space

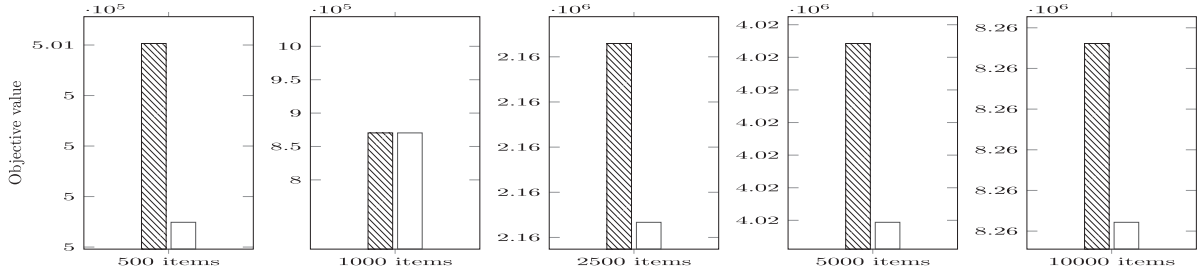

Fig. 5. Objective value of TSC algorithm without duplication (hatched) compared to TSC algorithm with duplication (unhatched).

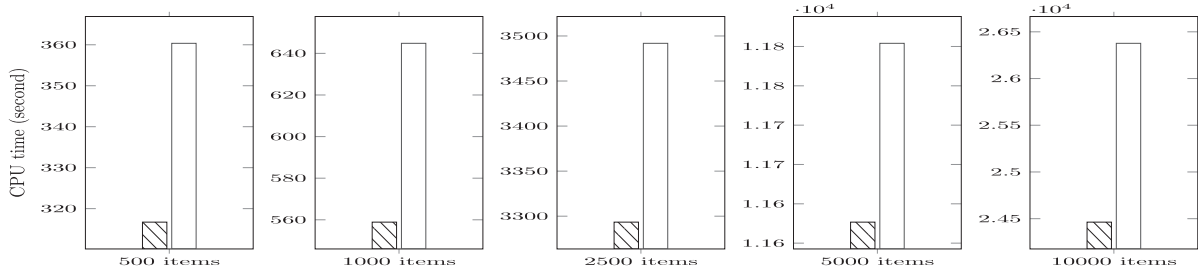

Fig. 6. CPU time of TSC algorithm without duplication (hatched) compared to TSC algorithm with duplication (unhatched).

by generating compound moves in a tree search fashion. An important aspect of carrying out the TSC processes is to avoid re-constructing already generated solutions. In order to avoid duplication, we adopt a new technique that makes a bijection between a KPS solution and an integer index. This technique proved efficient particularly in terms of solution quality and computation time. Our method was tested on a large set of randomly generated problems. The results showed that CPLEX was able to optimally solve only 39.5% of these problems; the rest had unknown optimal values. The experimental results showed that TSC produced good quality (optimal and near-optimal solutions) solutions in a short amount of time and allowed for the enhancement of the solution provided by CPLEX in 15 instances. Considering the promising performance of the TSC method presented in this work, further studies, some of which are currently underway in our laboratory, are needed to further extend the use of the space reduction technique to other general and critical problems.

References

Kong, X., Gao, L., Ouyang, H., & Li, S. (2015). A simplified binary harmony search algorithm for large scale 0–1 knapsack problems. *Expert Systems with Applications, 42*(12), 5337–5355.

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

Kellerer, H., Pferschy, U., & Pisinger, D. (2004). *Knapsack problem*. Berlin, Heidelberg: Springer-Verlag.

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

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

Yang, Y. (2006). *Knapsack problems with setup* Dissertation: Auburn University (August 7).

Mclay, A., & Jacobson, H. (2007). Algorithms for the bounded set-up knapsack problem. *Discrete Optimization, 4*, 206–212.

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

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

Glover, F. (1998). A template for scatter search and path relinking. In J.-K. Hao, E. Lutton, E. Ronald, M. Schoenauer, & D. Snyers (Eds.), *Artificial evolution lecture notes in computer science* (pp. 3–51) (1363).

Glover, F., & Laguna, M. (1999). *TABU search* : . Kluwer.

Rego, C., & Glover, F. (2002). Local search and metaheuristics for the travelling salesman problem. In G. Gutin & A. Punnen (Eds.) (pp. 309–368).

Greistorfer, P., & Rego, C. (2006). A simple filter-and-fan approach to the facility location problem. *Computers and Operations Research, 33*(9), 2590–2601.

Rego, C., Li, H., & Glover, F. (2011). A filter-and-fan approach to the 2d hp model of the protein folding problem. *Annals of Operations Research, 188*(1), 389–414.

Rego, C., & Duarte, R. (2009). A filter-and-fan approach to the job shop scheduling problem. *European Journal of Operational Research, 194*(3), 650–662.

Rego, C., & Mathew, F. (2011). A filter-and-fan algorithm for the capacitated minimum spanning tree problem. *Computers and Industrial Engineering, 60*, 187–194.

Khemakhem, M., Haddar, B., Chebil, K., & Hanafi, S. (2012). A filter-and-fan metaheuristic for the 0-1 multidimensional knapsack problem. *International Journal of Applied Metaheuristic Computing, 3*(4), 43–63.