Multistart solution-based tabu search for the Set-Union Knapsack Problem

Zequn Wei [a] and Jin-Kao Hao [a,*]

[a] *LERIA, Université d'Angers, 2 Boulevard Lavoisier, 49045 Angers, France*
Applied Soft Computing, 2021
https://doi.org/10.1016/j.asoc.2021.107260

Abstract

The NP-hard Set-Union Knapsack Problem is a general model able to formulate a number of practical problems. As a variant of the popular knapsack problem, SUKP is to find a subset of candidate items (an item is composed of several distinct weighted elements) such that a profit function is maximized while a knapsack capacity constraint is satisfied. We investigate for the first time a multistart solution-based tabu search algorithm for solving the problem. The proposed algorithm combines a solution-based tabu search procedure with a multistart strategy to ensure an effective examination of candidate solutions. We report computational results on 60 benchmark instances from the literature, including new best results (improved lower bounds) for 7 large instances. We show additional experiments to shed lights on the roles of the key composing ingredients of the algorithm. The code of the algorithm will be publicly available.

Keywords: Knapsack problems; Solution-based tabu search; Heuristics and metaheuristics; Combinatorial optimization.

1 Introduction

Given a set of elements $U = \{1, \ldots, n\}$, a set of items $V = \{1, \ldots, m\}$, each element has a weight $w_j > 0$ and each item has a profit $p_i > 0$. The items and elements are associated by a relation matrix $R_{ij}[m \times n]$ such that each item i corresponds to a subset of elements $U_i \subseteq U$. Let C be the capacity of a given knapsack. Then the Set-Union Knapsack Problem (SUKP) is to select

* Corresponding author.
Email addresses: zequn.wei@gmail.com (Zequn Wei), jin-kao.hao@univ-angers.fr (Jin-Kao Hao).

a subset of items S from V such that the total profit of S is maximized, while the total weight of the covered elements does not exceed the knapsack capacity C. In SUKP, the weight of an element is countered only one time even if the element appears in multiple selected items.

Formally, SUKP can be stated as follows.

$$(SUKP) \qquad \text{Maximize} \quad f(S) = \sum_{i \in S} p_i \tag{1}$$

$$\text{subject to} \quad W(S) = \sum_{j \in \cup_{i \in S} U_i} w_j \leq C, \; S \subseteq V \tag{2}$$

SUKP is known to be a useful model able to formulate a number of significant practical applications. For instance, consider the following project investing scenario where a company plans to invest in the purchase of machines for the production of new products. Each candidate product has a profit, while the composing parts of the product needs new machines to be acquired. The goal is to determine the set of products to be produced such that the profit will be maximized while ensuring that the cost of the purchased machines does not exceed a given budget. This problem can be conveniently formulated by the SUKP model where an item corresponds to a product with its profit and an element is a machine with its purchasing cost (element weight). Then, solving the project investing problem is equivalent to finding the optimal solution to the resulting SUKP problem. Generally, SUKP appears in the context of flexible manufacturing [1], financial decision making [2], applied cryptography [3], database design [4], data allocation in cyber systems [5] and key-pose caching systems [6]. However, SUKP is computationally difficult given that it belongs to the class of NP-hard problems [1].

Due to its relevance, SUKP has been investigated since the eighties and has received increasing attention in recent years. The existing approaches for solving SUKP can be classified into three families.

- *Exact and approximation algorithms*: These algorithms are theoretically able to find the optimal solutions or solutions of guaranteed quality. For instance, theoretical algorithms based on dynamic programming and greedy approximation were described in [7,1,8], which have an exponential complexity in the general case. Linear integer programming was investigated in [9], indicating that only small instances (with 85 to 100 items and elements) can be solved optimally by the CPLEX solver.
- *Population-based hybrid algorithms*: These algorithms are based on various bio-inspired metaheuristics operating with a population of solutions and associated search operators. In 2018, He et al. devised the first binary artificial bee colony algorithm for solving SUKP and provided a set of 30 benchmark instances (with 85 to 500 items and

elements) [10]. Since then, several other population algorithms were proposed, such as the binary swarm intelligence algorithm [11] (2019), weighted superposition attraction algorithm [12] (2018), swarm intelligence algorithm [13] (2019), group theory-based optimization algorithm [14] (2018), moth search algorithms [15,16] (2019), hybrid Jaya algorithm [17] (2020), estimation of distribution algorithm based on Lévy flight [18] (2020) and binary grey wolf optimization algorithm [19] (2020). In terms of computational performances, these approaches achieved interesting results. However, these algorithms are rather complex in design and most of them solve the binary SUKP problem indirectly by searching a continuous space.

- *Local search algorithms*: These algorithms are based on stochastic local search [20]. Contrary to the above population algorithms, local search algorithms solve the binary SUKP problem directly by examining candidate solutions in a discrete search space. In 2019, Wei and Hao [9] presented an iterated two-phase local search algorithm for SUKP, which includes a local optima exploration phase (using variable neighborhood descent and tabu search) and a local optima escaping phase to promote both intensification and diversification. This algorithm reported remarkable results on the 30 instances of [10]. Also in 2019, Lin et al. [21] introduced a local search (tabu search) procedure into the binary particle swarm optimization framework and achieved high-quality results on the 30 instances. In 2020, Wei and Hao [22] presented an effective kernel based tabu search algorithm and introduced a new set of 30 large instances for SUKP (with 585 to 1000 items and elements). The tabu search algorithms of [21,9,22] adopted the *attributed-based* tabu search method to avoid revisiting previous encountered solutions. During the search, the items involved in the move operations are recorded in a so-called tabu list and are excluded from consideration during a period called the tabu tenure. Computational results indicated that the local search approaches represent the current state-of-the-art in the literature in terms of solution quality and computational efficiency.

The tabu search technology [23] has been successfully applied to solve many difficult optimization problems. Although most studies rely on the popular and well-known *attributed-based* tabu search (ABTS) as exemplified by the studies of [21,22,24,25,26], recent studies indicated that the *solution-based* tabu search (SBTS) [27,28] is a highly competitive approach for solving several notoriously difficult binary optimization problems such as 0/1 multidimensional knapsack [29], multidemand multidimensional knapsack [30], minimum differential dispersion [31], and maximum min-sum dispersion [32] and obnoxious p-median [33]. Compared to the ABTS method, SBTS has the advantage of avoiding the use of tabu tenure and simplifying the determination of tabu status. Moreover, the intensification ability of SBTS tends to be stronger than that of ABTS. In addition, the study reported in

[30] on SBTS and our study (see Section 4.3) reveal that SBTS is more suitable than ABTS for solving a number of binary optimization problems. However, SBTS requires more resources (to record all the encountered solutions) than ABTS. More information on the SBTS approach can be found in recent studies such as [30,31,32,33], while some interesting studies using ABTS are provided in [25,24,34,35,26]. The main contributions of this work are summarized as follows.

To the best of our knowledge, no study has been reported in the literature investigating the interest of the SBTS approach for solving SUKP. In this work, we fill the gap by introducing the first multistart solution-based tabu search algorithm (MSBTS) for SUKP and provide additional indications of the benefits of the SBTS approach for binary optimization.

First, the proposed MSBTS algorithm integrates a dedicated solution-based tabu search approach and a multistart mechanism to ensure an effective and efficient examination of candidate solutions. During the search, each visited solution is recorded in a tabu list implemented with the help of a hash function based method such that the tabu status of a candidate solution can be easily determined in constant time. The multistart mechanism is employed to escape local optima traps. The algorithm is simple in design and frees the user from the delicate task of calibrating parameters. Second, we report new best-known results (improved lower bounds) for 7 large instances, which are useful for future research on SUKP. Third, we will make the code of our algorithm publicly available, which can be used by researchers and practitioners to solve various problems that can be formulated by the SUKP model.

The rest of the paper is structured as follows. In Section 2, we describe the general solution approach of the proposed algorithm and its main components. Section 3 is devoted to the performance assessment and comparisons with state-of-the-art algorithms. We analyze in Section 4 the influences of important components of the algorithm, followed by conclusions in the last section.

2 Multistart solution-based tabu search for the SUKP

2.1 Search space, solution representation, and evaluation function

Given a SUKP instance composed of m items, n elements, and knapsack capacity C, the proposed MSBTS algorithm explores the feasible search space Ω^F which includes all feasible candidate solutions corresponding to non-empty subsets of items satisfying the knapsack constraint, i.e.,

$$\Omega^F = \{y \in \{0,1\}^m : \sum_{j \in U_i} w_j \leq C, U_i = \{i : y_i = 1\}, 1 \leq i \leq m, 1 \leq j \leq n\} \tag{3}$$

Thus, a candidate solution S in Ω^F can be expressed by a m-dimensional binary vector $S = (y_1, \ldots, y_m)$, where y_i takes 1 if item i is selected, and 0 otherwise. Let $A = \{q : y_q = 1 \; in \; S\}$ and $\bar{A} = \{p : y_p = 0 \; in \; S\}$, a candidate solution can be equivalently represented by $S =< A, \bar{A} >$.

Additionally, the quality of a candidate solution S is determined by the objective function value $f(S)$ (Equation 1) of SUKP. Since SUKP is a maximization problem, a larger f value indicates a better solution.

2.2 Main framework

The MSBTS algorithm follows the flow chart shown in Fig. 1 and is described in Algorithm 1.

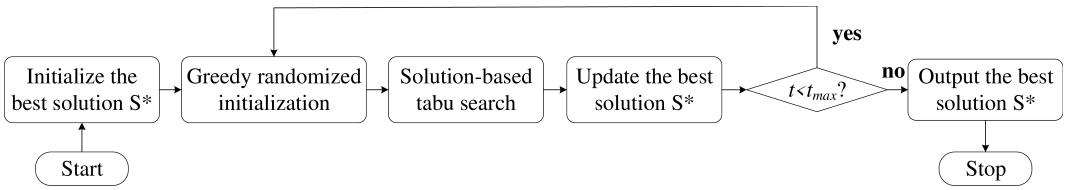

Fig. 1. Flow chart of the proposed MSBTS algorithm.

The basic idea of the MSBTS algorithm (see Alg. 1) is to repeat a greedy randomized initialization procedure (Section 2.3) followed by a solution-based tabu search procedure (Section 2.4). Specifically, after initializing the overall best solution S^* (line 3), the algorithm performs a 'while' loop (lines 4-10) to execute the main search process. At each round of this process, MSBTS first runs the greedy randomized procedure to generate a starting solution S (line 5), which is used as the input solution of the solution-based tabu search procedure. The solution-based tabu search procedure (line 6) iteratively improves the input solution S and returns the local best solution S_b encountered. After conditionally updating the overall best solution S^*, the algorithm moves to the next round of its search by re-starting the greedy randomized procedure. The main search process is terminated and returns the overall best solution S^*, when the cut-off time (t_{max}) is reached.

Algorithm 1 Multistart solution-based tabu search for the SUKP

Input: Instance I, cut-off time t_{max}, neighborhoods N, hash vectors H_1, H_2, H_3, length of hash vectors L, hash functions h_1 , h_2 , h_3.
Output: The best solution found S^*.
$S^* \leftarrow \emptyset$ /* Initialize the overall best solution S^* (i.e., $f(S^*) = 0$)*/
while $Time \leq t_{max}$ **do**
  $S \leftarrow Greedy\_Randomized\_Initialization(I)$
  /* Record the best solution S_b found during tabu search */
  $S_b \leftarrow Solution\_Based\_Tabu\_search(S)$
  if $f(S_b) > f(S^*)$ **then**
    $S^* \leftarrow S_b$ /* Update the overall best solution S^* found so far */
  end if
end while
return S^*

2.3 Greedy randomized initialization

The quality of initial solutions may impact the performance of the algorithm. In this work, we adopt a greedy randomized initialization procedure to generate initial solutions of good quality.

Let $W(S)$ be the total weight of the current solution S and W_k be the additional weight of a non-selected item k, where W_k is defined by $W_k = \sum_{j \in U_k \wedge j \notin \cup_{i \in S} U_i} w_j$. Then the feasible non-selected items can be expressed by $R(x) = \{k \in \bar{A} : W_k + W(S) \leq C\}$, where $\bar{A}$ is the set of non-selected items. Following [36], we employ a restricted candidate list (denoted by RCL) to record rcl feasible non-selected items belonging to $R(x)$, where rcl is the maximum size of RCL. A too large rcl value will make many items to be recorded in RCL and thus result in an initial solution of poor quality, while a too small rcl value will limit the possible choices and lead to insufficient diversity of the initialization procedure. In our case, we set empirically $rcl = \sqrt{max\{m, n\}}$, where m and n are the number of items and elements respectively. Considering the fact that the number of items in $R(x)$ may be less than rcl, we finally set the size of RCL by $|RCL| = min\{rcl, |R(x)|\}$. Now, we build the restricted candidate list as follows. For each item k of $R(x)$, we calculate its dynamic profit ratio $r_k^* = p_k/W_k$. Then we identify the top $|RCL|$ items with the largest r^* values to form RCL. As the result, RCL contains the feasible non-selected items whose dynamic profit ratio is larger than the other non-selected items. Finally, each item k in RCL is selected with probability P_k, which is given by $P_k = r_k^* / \sum_{l=1}^{|RCL|} r_l^*$.

As shown in Algorithm 2, starting from an empty solution S, the initialization procedure randomly and adaptively adds feasible items k into S

Algorithm 2 Greedy Randomized Initialization

1: **Input**: Instance I.
2: **Output**: The initial solution S.
3: /* Get the knapsack capacity C and restricted candidate list length rcl */
$(C, rcl) \leftarrow Read\_instance(I)$
4: $W(S) \leftarrow 0$ /* Initialize the total weight of S */
5: **while** $W(S) \leq C$ **do**
6: Calculate additional weight W_k of each non-selected item k
7: Add all items i with $W_i = 0$ into current solution S
8: $r^* \leftarrow Calculate\_dynamic\_profit\_ratio(W, |RCL|)$
9: $P \leftarrow Calculate\_probability(r^*, |RCL|)$
10: $S \leftarrow Add\_one\_item(P, S)$
11: **end while**
12: **return** S

at each iteration of the 'while' loop (lines 5-11). Specifically, the initial solution is generated by four steps. First, we calculate the additional weight W_k of each non-selected item k (line 6), and add all items k with $W_k = 0$ into the current solution S, which means adding this item will not increase the total weight of S (lines 7). Second, we calculate the dynamic profit ratio r_k^* of each item k in $R(x)$ with $W_k \neq 0$ (line 8). Third, we calculate the selection probability P_k of each item k (line 9). Fourth, we randomly add one item from RCL into S according to P_k (line 10). These four steps are repeated until the knapsack capacity is reached.

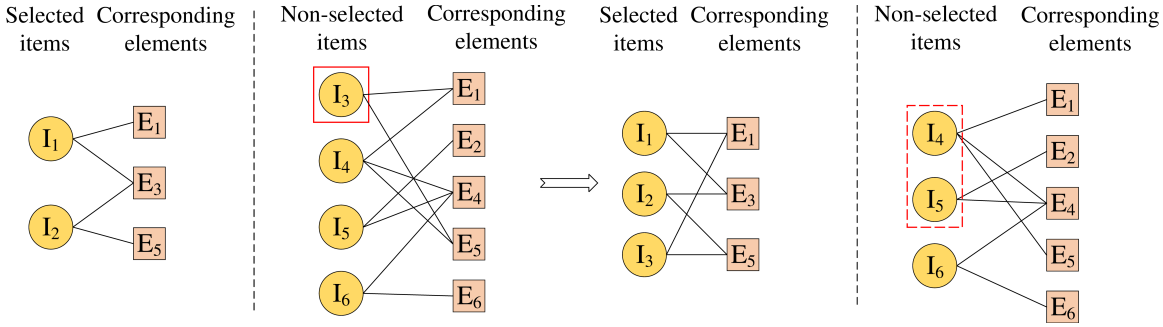

Fig. 2. An illustrative example of the main steps of the greedy randomized initialization procedure.

As shown in Fig. 2, we present a numerical example to illustrate the main steps of the greedy randomized initialization procedure. Given a set of six items (I_i, $i = 1, \ldots, 6$) with a profit of 1 to 6 respectively and a set of 6 elements (E_j, $j = 1, \ldots, 6$) with a weight of 1 to 6 respectively. Let the capacity of knapsack be equal to 16. At the step shown in the left figure, two items I_1 and I_2 are already added into the knapsack. We calculate additional weight W_i of each non-selected item i and find that $W_3 = 0$ (the elements E_1 and E_5 corresponding to item I_3 are already selected). Then we add the item I_3 into the knapsack and obtain the new solution shown in the right figure. Next, we

calculate the dynamic profit ratio of the non-selected items and identify items I_4 and I_5 as belonging to RCL (in this case, $|RCL| = 2$). Finally, we add one of the two items into the knapsack according to the probability P_k.

2.4 Solution-based tabu search

Tabu search (TS) is a general and powerful metaheuristic for combinatorial optimization [23]. Typically, TS examines candidate solutions by iteratively transitioning from the current solution to a nearby (neighbor) solution by following a neighborhood. Each solution transition is performed by selecting the best admissible candidate among the neighboring solutions within the neighborhood. The key distinguishing feature of TS compared to other local optimization approaches is its tabu list strategy, which prevents the search from revisiting previously encountered solutions. With the so-called *solution-based* tabu search [27,28], the tabu list is implemented with hash vectors and associated hash functions. Contrary to the popular *attribute-based* tabu search approach which typically needs some parameters for tabu list management, solution-based tabu search has the advantage of eliminating such parameters.

In the context of solving SUKP, the best-performing algorithms are all based on the conventional attribute-based TS approach [21,9,22]. This work adopts for the first time the solution-based tabu search approach for solving SUKP, which leads to an effective algorithm while avoiding the difficulty of tuning parameters.

Algorithm 3 shows the general scheme of our solution-based tabu search (SBTS) procedure. After initializing the best solution found so far (line 3) and the associated hash vectors (i.e., tabu list, line 4), the SBTS procedure iteratively improves the current solution S (lines 6-20) until 1) no admissible neighboring solution (i.e., feasible and non-tabu neighboring solution) exists, or 2) the allowed cut-off time t_{max} is reached. Given the optimization function f, the neighborhood structure N (Section 2.4.1) and the tabu list management strategy (Section 2.4.2), the current solution S is replaced by a best admissible neighboring solution at each iteration of the SBTS procedure. And then the tabu list is updated with the newly obtained solution S. The best solution found during this procedure is recorded in S_b (lines 14-16) and returned as the output of SBTS. Note that the best admissible neighboring solution S is not necessarily better than S_b, but it will still be selected to replace the current solution S. In this way, the search can keep moving forward to discover better solutions without being trapped in local optima.

The SBTS procedure terminates under one of the two following conditions: (1)

the overall cut-off time is reached; (2) no admissible neighboring solution can be found in the neighborhood, i.e., $N'(S) = \varnothing$ where $N'(S) \subseteq N(S)$ is the set of the admissible neighboring solutions not forbidden by the tabu list. Upon the termination of the SBTS procedure, two cases are considered: the overall cut-off time is reached and then the whole algorithm terminates. Otherwise, the algorithm re-starts its search by using the greedy randomized initialization procedure to creating a new starting solution, which is used to seed the next round of the SBTS procedure.

Next, we present the main ingredients of SBTS, including the move operator, the neighborhood structure and the tabu list strategy.

Algorithm 3 Solution-based tabu search

Input: Input solution S, neighborhood N, hash vectors H_1, H_2, H_3, hash functions h_1, h_2, h_3, cut-off time t_{max}, length of hash vectors L.
Output: Best solution S_b found during tabu search.
$S_b \leftarrow S$ /* Record the best solution S_b found during tabu search */
$(H_1, H_2, H_3) \leftarrow Initialize\_Hash\_Vectors(H_1, H_2, H_3, L)$ /* (i.e., tabu list) */
$Find \leftarrow True$ /* Track the admissible neighboring solution */
while $Find \wedge Time \leq t_{max}$ **do**
  Find admissible neighboring solutions $N'(S)$ in $N(S)$
  if $N'(S) \neq \varnothing$ **then**
    /* Attain the best admissible neighboring solution S */
    $S \leftarrow argmax\{f(S') : S' \in N'(S)\}$
    $Find \leftarrow True$
  else
    $Find \leftarrow False$
  end if
  if $f(S) > f(S_b)$ **then**
    $f(S_b) \leftarrow f(S)$ /* Update the best solution S_b found during tabu search */
  end if
  /* Update the hash vectors with S */
  $H_1[h_1(S)] \leftarrow 1$
  $H_2[h_2(S)] \leftarrow 1$
  $H_3[h_3(S)] \leftarrow 1$
end while
return S_b

2.4.1 Move operator and neighborhood structure

Our SBTS procedure relies on two popular move operators, i.e., the $flip$ operator and the *swap* operator to explore candidate solutions. Specifically, given a solution $S = (y_1, \ldots, y_m)$ as described in Section 2.1, the $flip(i)$ operator changes the value of a variable y_i to its opposite value $1 - y_i$. Similarly, given a solution $S =< A, \bar{A} >$, the $swap(q, p)$ operator exchanges one item in A against one item in $\bar{A}$, where q and p represent items in sets A

and $\bar{A}$ respectively. Meanwhile, a neighborhood filtering strategy [9,22] is applied in both move operators to reduce the neighborhood size. So the neighborhoods $N_f(S)$ and $N_s(S)$ induced by $flip(i)$ and $swap(q,p)$ are defined as follows, respectively.

$$N_f(S) = \{S' : S' = S \oplus flip(i) : 1 \leq i \leq m, f(S') > f(S_b)\} \tag{4}$$

$$N_s(S) = \{S' : S' = S \oplus swap(q,p) : q \in A, p \in \bar{A}, f(S') > f(S_b)\} \tag{5}$$

In this work, we employ a union neighborhood that covers both neighborhoods $N_f(S)$ and $N_s(S)$, i.e., $N(S) = N_f(S) \cup N_s(S)$. Moreover, we also apply a streamlining *gain updating strategy* to quickly evaluate the weight of each neighboring solution (see [21,22] for more details).

2.4.2 Tabu list management strategy using hash functions

During the SBTS procedure, the current solution S is iteratively replaced by the best admissible neighboring solution S', which is identified according to the objective function value and the tabu list strategy described in this section. Unlike the traditional attribute-based tabu search, where the tabu list records the performed moves, our solution-based tabu search uses hash vectors and hash functions to implement the tabu list.

Following previous studies [29,30,31,32], our tabu list management strategy relies multiple hash vectors and hash functions, which helps significantly reduce the probability of wrong identification of the tabu status. Specifically, we adopt three hash vectors H_v $(v = 1, 2, 3)$ of length L, where each position takes a binary value which contributes to the definition of the tabu status of candidate solutions. The hash vectors are initialized to 0, indicating that no candidate solution is classified as tabu. Once a candidate solution is selected to replace the current solution S, the corresponding positions in the three hash vectors will be set to 1 (i.e., $H_v[h_v(S)] \leftarrow 1$, $v = 1, 2, 3$).

Given a candidate solution $S = (y_1, \ldots, y_m)$ where $y_i = 1$ if item i is selected, and $y_i = 0$ otherwise, the hash values $h_v(S)$ $(v = 1, 2, 3)$ are calculated by

$$h_v(S) = (\sum_{i=1}^{m} \lfloor \mathcal{W}_i^v \times y_i \rfloor) \bmod L \tag{6}$$

where L is the length of the hash vectors and is set to 10^8. And $\mathcal{W}_i^v$ is a pre-computed weight that satisfies the following relation: $\mathcal{W}_i^v = i^{\gamma_v}$ ($v = 1, 2, 3$ and $i = 1, \ldots, m$), where γ_v is a parameter that takes different values for the three

hash functions ($\gamma_v = 1.2, 1.6, 2.0$). To reduce the possible collisions that occur with hash functions, we randomly shuffle the order in the pre-computed weight vector $\mathcal{W}^v$ in order to ensure an extended distribution of hash values of the solutions. Fig. 3 shows an illustrative example of this shuffling operation with five items and γ_v being set to $1.2, 1.6, 2.0$, respectively. The left figure indicates the pre-computed weights $\mathcal{W}_i^v$ ($v = 1, 2, 3$, and $i = 1, \ldots, 5$). Then the order of each of the three weight vectors $\mathcal{W}^v$ is randomly shuffled to obtain a new weight vector shown in the right figure. Our preliminary experiment indicates that this random shuffling operation helps to reduce the error rates of the hash functions. We present the rationale for the setting of γ_v and an analysis of the hash functions in Section 4.1.

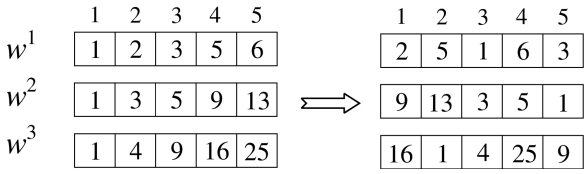

Fig. 3. An illustrative example of the random shuffling operation.

The hash-based tabu list management strategy works as follows. Given a candidate solution $S = (y_1, \ldots, y_m)$, we first calculate the three hash values $h_v(S)$ that are the indexes of the hash vectors. Then, the tabu status of solution S is determined according to the values of the hash vectors $H_v[h_v(S)]$. Specifically, S is determined as a forbidden solution (i.e., already visited) when $H_1[h_1(S)] \wedge H_2[h_2(S)] \wedge H_3[h_3(S)] = 1$. Otherwise, S is classified as an unforbidden solution that has not been visited by this round of SBTS and is eligible for solution transition. In this way, we can quickly determine the tabu status of a neighboring solution in $O(1)$, and this is the main advantage of the hash-based tabu list management strategy. For the illustrative example shown in Fig. 4, solution S is classified as tabu and thus is excluded for solution transition.

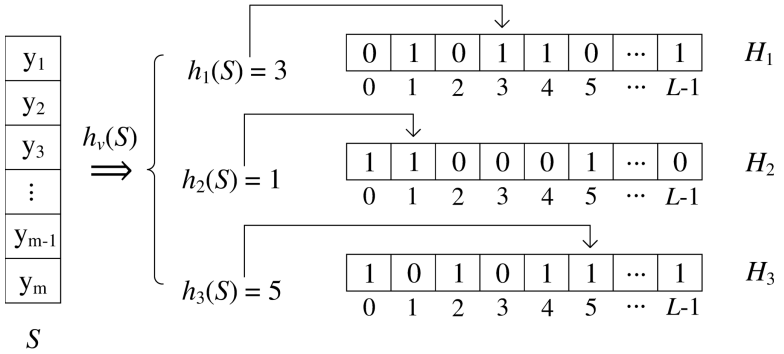

Fig. 4. An example of a solution forbidden by the hash functions and the associated hash vectors.

From an empty subset S, the greedy randomized initialization procedure (Section 2.3 and Algorithm 2) creates a solution in four steps. The first step calculates the additional weights in $O(m \times n)$, where m is the number of items and n is the number of elements. The second step calculates the dynamic profit ratios and identifies the top items in $O(m \times log(m))$. The third step of calculating probability can be realized in $O(|RCL|)$ and the fourth step of adding one item can be achieved in $O(1)$. Then the time complexity of the initialization procedure is $O(m \times n \times K_1)$, where K_1 is the maximum iterations of the initialization procedure. For the main solution-based tabu search procedure (Section 2.4 and Algorithm 3), we can evaluate its complexity as follows. Let $S =< A, \bar{A} >$ be a given input solution, the complexity of one iteration of the SBTS procedure is $O((m + |\bar{A}| \times |A|) \times n)$. Let K_2 be the maximum iterations of SBTS. Then the time complexity of SBTS is $O((m + |\bar{A}| \times |A|) \times n \times K_2)$.

Now we discuss the relations between our algorithm and the existing tabu search algorithms for SUKP [21,9,22]. First, MSBTS is the first solution-based tabu search algorithm for SUKP, while the existing TS algorithms are based on the conventional attribute-based TS approach. Second, MSBTS employs a new tabu list management strategy that avoids tuning the tabu tenure. Third, unlike the previous TS algorithms that uses a perturbation procedure, MSBTS does not need such specific diversification strategies. Yet, it achieves remarkable results, as it is shown in Section 3.

Finally, it is worth mentioning that the solution-based tabu search approach has led to highly effective algorithms for several NP-hard binary problems such as 0-1 multidimensional knapsack [29], multidemand multidimensional knapsack [30], minimum differential dispersion [31] and maximum min-sum dispersion [32] and obnoxious p-median [33]. Our study of using solution-based tabu search for SUKP further confirms the usefulness of this approach for binary optimization.

3 Computational results and comparisons

This section is devoted to a computational assessment of the proposed MSBTS algorithm, in comparison with three best-performing SUKP algorithms in the literature based on two sets of 60 benchmark instances available at `http://www.info.univ-angers.fr/pub/hao/SUKP_MSBTS.html`.

3.1 Benchmark instances

The SUKP benchmark instances adopted in our experiments were commonly tested in the literature, which can be divided into Set I and Set II. The Set I instances were proposed in [10] with 85 to 500 items and elements, while the Set II instances were introduced in [22] with 585 to 1000 items and elements. These 60 instances share the same characteristics. An instance is defined by m items, n elements and an associated binary relation matrix $R_{ij}[m \times n]$, where $R_{ij} = 1$ means that item i contains element j. Each instance is further characterized by two parameters: the density α of $R_{ij} = 1$ in the relation matrix R (i.e., $\alpha = (\sum_{i=1}^{m} \sum_{j=1}^{n} R_{ij})/(mn)$) and the ratio β of knapsack capacity C to the total weight of the elements (i.e., $\beta = C/\sum_{j=1}^{n} w_j$). As indicated in [10,22], for the 60 instances tested in this study, α is equal to 0.10 or 0.15, while β is equal to 0.75 or 0.85.

3.2 Experimental settings

The proposed MSBTS algorithm was implemented in C++ and compiled using the g++ compiler with the -O3 option. All the experiments were carried out on an Intel Xeon E5-2670 processor (2.5 GHz CPU and 2 GB RAM) running under the Linux operating system. The MSBTS algorithm used the same stopping conditions for the reference algorithms (see below), i.e., 500 seconds for the Set I instances and 1000 seconds for the Set II instances. Each instance was solved 100 times independently with different random seeds. Note that contrary to the existing algorithms, our algorithm eliminates the need for tuning parameters.

Among the existing algorithms for SUKP in the literature, we identify four best performing algorithms according to the reported computational results: hybrid jaya algorithm [17] (DHJaya, 2019), hybrid binary particle swarm optimization with tabu search algorithm [21] (HBPSO/TS, 2019), iterated two-phase local search algorithm [9] (I2PLS, 2019) and the kernel based tabu search algorithm [22] (KBTS, 2020). We thus use them as the reference algorithms for our comparative study. Since the results of these algorithms were obtained in [22] on the same computing platform and under the same stopping condition as in this work, we directly adopt these results in our study.

3.3 Computational results

The computational results of our MSBTS algorithm and the reference algorithms on the SUKP instances of Set I and Set II are reported in Table 1

and 2, respectively [1]. The first column of these two tables gives the name of each instance, where the asterisk (*) denotes that the optimal value proved by CPLEX [9]. The remaining columns report the following information: the best objective value (f_{best}), the average objective value (f_{avg}), the standard deviations over 100 runs (std) and the average run times t_{avg} (to obtain the f_{best} value) of each involved algorithm. The row #Avg shows the average value of each column. Furthermore, the **bold** entries highlight the dominating values among the compared results, while the *italic* entries indicate the equal best values.

Comparing the results of Table 1 leads to the following comments. First, in terms of the best performance indicator, MSBTS can attain all the best-known f_{best} results on all the 30 instances of Set I, thus dominating DHJaya and matching the performance of the best algorithms I2PLS and KBTS. Second, in terms of the average performance indicator, our MSBTS algorithm dominates DHJaya and competes favorably with HBPSO/TS and I2PLS, while performing marginally worse than KBTS even if MSBTS has better f_{avg} results on five large instances with 485 to 500 items and elements. It is difficult to further compare the competing algorithms on Set I, since the *p-values* in Table 3 from the non-parametric Wilcoxon signed-rank test don't show a statistical difference at 0.05 significance level between MSBTS and the reference algorithms except DHJaya. So we focus on Set II for a more detailed comparison.

Table 2 on the 30 instances of Set II discloses that our MSBTS algorithm outperforms the reference algorithms on large size instances. Specifically, MSBTS matches the best-known f_{best} values for the remaining 23 instances, and remarkably, finds 7 new best-known results (improved lower bounds). Most of these 7 instances have 985 to 1000 items, which demonstrates the advantage of our algorithm on the most difficult instances. When considering the average performance, MSBTS remains highly competitive compared to the reference algorithms. On the other hand, MSBTS has a zero std value on 20 instances while the reference algorithms achieve less zero std values (0 for DHJaya, 1 for HBPSO/TS, 2 for I2PLS and 6 for KBTS), which shows the robustness of our algorithm. Moreover, the smallest #Avg value of the corresponding t_{avg} entries obtained by MSBTS demonstrates that our algorithm is more computational efficient than the reference algorithms on this set of SUKP instances. We show a detailed time-efficiency comparison of our MSBTS algorithm with the reference algorithms in Section 3.4.

In order to better highlight the advantage of the proposed MSBTS algorithm,

[1] Our solution certificates are available at: `http://www.info.univ-angers.fr/pub/hao/SUKP_MSBTS.html`. The code of MSBTS will also be made available upon the publication of the paper.

we summarize the comparative results between MSBTS and each reference algorithm in Table 3. The first two columns of the table give the pairs of two compared algorithms and the corresponding instance sets, respectively. Columns #Wins, #Ties and #Losses show the number of instances for which MSBTS obtains a better, equal and worse result according to the f_{best} and f_{avg} indicators. The last column indicates the *p-values* from the Wilcoxon signed-rank test, where 'NA' implies that two underlying groups of results are exactly the same. From Table 3, we can observe that MSBTS achieves better or equal results in terms of f_{best} on all the tested instances, while being better in terms of f_{avg} on most instances. Note that KBTS reports more f_{avg} values better than MSBTS for Set I (13 vs 6). However, the Wilcoxon signed-rank test in Table 3 ($p\text{-}value$ = 9.10e-2 $>$ 0.05) indicates that there is no statistically significant difference. Furthermore, as shown in the last column, the *p-values* (< 0.05) obtained between MSBTS and each compared algorithm on the instances of Set II confirm the statistically significant difference of the compared results.

3.4 Time-to-target analysis

We now present a time-to-target analysis (TTT) to evaluate the computational efficiency of the proposed MSBTS algorithm compared to the reference algorithms. For this, we compare the time required for each algorithm to obtain a solution at least as good as a given target value and measure the empirical probability distributions. More details about TTT can be found in [37,38].

Specifically, we run each compared algorithm 100 times to solve each instance of Set II with the setting shown in Section 3.2 and recorded the time to achieve an objective value at least as good as the given target value (the algorithm stops immediately when it reaches the target value). Then we sorted the times in increasing order and calculated the probability $\rho_i = (i - 0.5)/100$ with each time T_i, where T_i corresponds to the ith smallest time.

Table 4 shows the experimental results of DHJaya, HBPSO/TS, I2PLS, KBTS and MSBTS on the instances of Set II. The first two columns give the name of each instance and the corresponding target value, respectively. The remaining columns report the best time (T_{best}) in seconds to achieve the target value and the average time (T_{avg}) in seconds to reach the target value over 100 runs. The row (#Avg) indicates the average value of each column. And the row #Best shows the number of instances for which an algorithm obtains the smallest T_{best} value among the compared algorithms. Moreover, to check whether there exists a significant difference between the proposed MSBTS algorithm and the compared algorithms in terms of T_{best} and T_{avg}, we report the *p-values* from

Table 1. Computational results of the MSBTS algorithm and the reference algorithms on the 30 benchmark instances of Set I.

Instance/Algorithm	DHJaya [17]				HBPSO/TS [21]				I2PLS [9]				KBTS [22]				MSBTS			
	f_{best}	f_{avg}	std	$t_{avg}(s)$	f_{best}	f_{avg}	std	$t_{avg}(s)$	f_{best}	f_{avg}	std	$t_{avg}(s)$	f_{best}	f_{avg}	std	$t_{avg}(s)$	f_{best}	f_{avg}	std	$t_{avg}(s)$
100_85_0.10_0.75*	*13283*	*13283*	0	9.477	*13283*	*13283*	0	0.098	*13283*	*13283*	0	3.094	*13283*	*13283*	0	4.082	*13283*	*13283*	0	12.770
100_85_0.15_0.85*	*12479*	*12479*	0	24.414	*12479*	12403.15	98.97	101.122	*12479*	12335.13	98.78	103.757	*12479*	*12479*	0	42.992	*12479*	12413.78	79.79	184.323
200_185_0.10_0.75	*13521*	13498.22	26.10	258.213	*13521*	*13521*	0	0.490	*13521*	*13521*	0	71.984	*13521*	*13521*	0	6.988	*13521*	*13521*	0	22.528
200_185_0.15_0.85	*14215*	**14215**	0	83.129	*14215*	14177.38	70.84	72.041	*14215*	14031.28	131.46	180.809	*14215*	14209.87	29.17	107.407	*14215*	13946.15	153.67	258.541
300_285_0.10_0.75	11385	11167.77	129.98	174.335	*11563*	*11563*	0	38.355	*11563*	11562.02	3.94	181.248	*11563*	*11563*	0	28.841	*11563*	*11563*	0	37.877
300_285_0.15_0.85	12402	12248.42	22.12	316.767	*12607*	**12607**	0	24.967	*12607*	12364.55	83.03	240.333	*12607*	12536.02	87.51	235.450	*12607*	12430.51	73.86	216.465
400_385_0.10_0.75	*11484*	11325.88	38.65	229.370	*11484*	*11484*	0	10.870	*11484*	*11484*	0	31.801	*11484*	*11484*	0	0.296	*11484*	*11484*	0	7.643
400_385_0.15_0.85	10710	10293.96	173.85	241.068	*11209*	*11209*	0	16.478	*11209*	11157.26	87.29	141.525	*11209*	*11209*	0	72.020	*11209*	*11209*	0	46.800
500_485_0.10_0.75	11722	11675.51	55.53	226.604	*11771*	11746.19	57.98	293.514	*11771*	11729.76	6.59	349.545	*11771*	11755.47	19.74	206.199	*11771*	**11771**	0	31.171
500_485_0.15_0.85	10194	9703.56	114.852	383.021	10194	10163.76	82.11	92.121	*10238*	10133.94	94.72	369.375	*10238*	10202.90	16.25	293.140	*10238*	**10205.62**	16.33	389.536
100_100_0.10_0.75*	*14044*	*14044*	0	1.374	*14044*	*14044*	0	0.518	*14044*	*14044*	0	38.245	*14044*	*14044*	0	0.023	*14044*	*14044*	0	6.639
100_100_0.15_0.85*	*13508*	*13508*	0	1.572	*13508*	*13508*	0	2.923	*13508*	13451.50	126.49	70.587	*13508*	*13508*	0	33.403	*13508*	*13508*	0	55.103
200_200_0.10_0.75	*12522*	12480.62	65.05	207.667	*12522*	*12522*	0	0.8125	*12522*	*12522*	0	54.780	*12522*	*12522*	0	48.206	*12522*	12518.28	21.15	70.411
200_200_0.15_0.85	*12317*	12217.81	93.361	229.824	*12317*	*12317*	0	0.950	*12317*	12280.07	57.77	238.348	*12317*	*12317*	0	72.495	*12317*	12316.21	7.86	92.155
300_300_0.10_0.75	12736	12676.78	35.20	241.774	*12817*	12806.44	15.39	29.074	*12817*	*12817*	0	66.403	*12817*	*12817*	0	74.247	*12817*	12813.70	9.90	165.618
300_300_0.15_0.85	11425	11260.25	103.95	152.329	*11585*	*11585*	0	5.985	*11585*	11512.18	73.15	220.100	*11585*	11584.17	8.26	141.464	*11585*	*11585*	0	156.333
400_400_0.10_0.75	11569	11301.56	74.88	322.143	*11665*	11484.20	72.95	45.025	*11665*	*11665*	0	18.733	*11665*	*11665*	0	64.126	*11665*	11657.08	10.56	90.423
400_400_0.15_0.85	10927	10721.45	221.38	77.037	*11325*	*11325*	0	5.902	*11325*	*11325*	0	76.000	*11325*	*11325*	0	17.591	*11325*	11309.20	112.46	125.950
500_500_0.10_0.75	10943	10871.22	39.93	41.383	11109	11026.24	51.62	340.958	*11249*	11243.40	27.43	134.186	*11249*	11248.96	0.40	146.040	*11249*	**11249**	0	29.905
500_500_0.15_0.85	10214	10069.33	103.33	101.926	*10381*	10213.25	71.30	220.328	*10381*	10293.89	85.53	237.894	*10381*	10362.63	52.25	156.331	*10381*	**10365.52**	49.41	169.084
85_100_0.10_0.75*	*12045*	*12045*	0	17.199	*12045*	*12045*	0	0.056	*12045*	*12045*	0	2.798	*12045*	*12045*	0	0.075	*12045*	*12045*	0	3.117
85_100_0.15_0.85*	*12369*	*12369*	0	0.342	*12369*	*12369*	0	0.088	*12369*	12315.53	62.60	17.470	*12369*	*12369*	0	10.175	*12369*	*12369*	0	26.240
185_200_0.10_0.75	*13696*	13667.63	26.56	244.205	*13696*	*13696*	0	0.489	*13696*	13695.60	3.68	124.136	*13696*	*13696*	0	5.851	*13696*	*13696*	0	7.089
185_200_0.15_0.85	*11298*	*11298*	0	38.439	*11298*	*11298*	0	0.486	*11298*	11276.17	83.78	139.865	*11298*	*11298*	0	6.373	*11298*	*11298*	0	30.689
285_300_0.10_0.75	*11568*	11563.80	10.41	203.874	*11568*	*11568*	0	13.630	*11568*	*11568*	0	25.128	*11568*	*11568*	0	30.618	*11568*	11567.70	2.99	17.706
285_300_0.15_0.85	11714	11436.93	101.85	463.466	*11802*	**11802**	0	2.135	*11802*	11790.43	27.51	206.422	*11802*	11799.27	9.95	168.904	*11802*	11798.88	10.58	186.685
385_400_0.10_0.75	10483	10287.36	80.61	53.459	*10600*	10552.73	74.68	100.155	*10600*	10536.53	56.08	234.475	*10600*	**10600**	0	73.087	*10600*	10599.70	1.71	150.505
385_400_0.15_0.85	10302	10184.09	138.00	230.077	*10506*	10472.40	67.20	168.870	*10506*	10502.64	23.52	129.505	*10506*	**10506**	0	58.240	*10506*	10504.23	16.08	133.340
485_500_0.10_0.75	11036	10883.19	48.58	66.029	*11321*	11142.27	62.51	223.387	*11321*	11306.47	36.00	207.118	*11321*	11318.81	10.95	121.494	*11321*	**11321**	0	54.178
485_500_0.15_0.85	10104	9665.70	142.57	49.438	*10220*	10208.96	3.26	143.999	*10220*	10179.45	46.97	238.630	*10220*	**10219.76**	1.68	118.564	*10220*	10219.04	3.26	123.052
#Avg	11873.83	11748.07	61.56	156.332	11967.47	11938.10	24.29	65.194	*11973.60*	11932.39	40.54	138.476	*11973.60*	**11968.56**	7.87	78.16	*11973.60*	11953.72	18.987	96.729

Table 2. Computational results of the MSBTS algorithm and the reference algorithms on the 30 benchmark instances of Set II.

Instance/Algorithm	DHJaya [17]				HBPSO/TS [21]				I2PLS [9]				KBTS [22]				MSBTS			
	f_{best}	f_{avg}	std	$t_{avg}(s)$	f_{best}	f_{avg}	std	$t_{avg}(s)$	f_{best}	f_{avg}	std	$t_{avg}(s)$	f_{best}	f_{avg}	std	$t_{avg}(s)$	f_{best}	f_{avg}	std	$t_{avg}(s)$
600_585_0.10_0.75	9640	9449.97	60.22	690.489	9741	9724.60	7.68	576.260	9750	9734.74	13.39	479.356	*9914*	*9914*	0	209.679	*9914*	*9914*	0	181.952
600_585_0.15_0.85	9187	8998.45	79.17	881.295	*9357*	9174.16	143.19	413.157	*9357*	9324.62	16.67	457.807	*9357*	9354.52	9.18	263.684	*9357*	**9357**	0	59.382
700_685_0.10_0.75	9790	9602	55.96	543.236	*9881*	9792.23	51.06	881.999	*9881*	9819.24	38.74	363.945	*9881*	9844.96	11.88	455.713	*9881*	**9881**	0	28.474
700_685_0.15_0.85	9106	8894.09	140.48	426.088	9135	8940.65	109.78	689.759	*9163*	9135.27	4.90	671.132	*9163*	9138.36	9.10	524.799	*9163*	**9163**	0	102.379
800_785_0.10_0.75	9771	9540.08	47.95	637.331	9837	9736.89	46.11	777.755	9822	9678.89	80.67	719.986	9837	9808.86	20.42	483.384	**9937**	**9937**	0	259.160
800_785_0.15_0.85	8797	8649	63.01	236.798	8907	8872.84	84.36	418.033	8907	8780.32	43.34	674.231	*9024*	8955.29	49.07	474.643	*9024*	**8986.25**	25.38	486.666
900_885_0.10_0.75	9455	9249.53	109.14	687.150	9611	9560.93	89.43	514.922	9611	9537.61	61.42	511.245	*9725*	9616.70	24.85	609.811	*9725*	**9725**	0	192.213
900_885_0.15_0.85	8418	8244.47	87.93	316.604	8481	8208.22	108.56	332.102	8481	8426.36	44.76	541.670	*8620*	8526.55	48.37	274.653	*8620*	**8566.71**	31.18	978.573
1000_985_0.10_0.75	9424	9306.86	45.01	309.873	9668	9278.50	125.80	620.436	9580	9221.23	103.18	329.743	9668	9496.63	74.35	487.925	**9689**	**9632.59**	29.56	671.192
1000_985_0.15_0.85	8433	8280.52	90.87	312.589	8448	8129.08	92.71	564.848	8448	8268.18	135.55	541.606	8453	8448.05	0.50	941.565	**8455**	**8453.36**	0.77	634.006
600_600_0.10_0.75	10507	10504.25	19.67	321.196	10518	10517.89	1.09	60.254	*10524*	10520.70	2.99	513.537	*10524*	10521.72	2.91	404.697	*10524*	**10524**	0	16.377
600_600_0.15_0.85	8910	8785.64	43.46	571.965	9024	8902.33	27.27	214.261	*9062*	9022.97	46.28	456.386	*9062*	9061.16	4.78	255.342	*9062*	**9062**	0	224.626
700_700_0.10_0.75	9512	9409.01	28.70	809.836	*9786*	9679.56	72.51	215.910	*9786*	9742.73	40.87	383.700	*9786*	*9786*	0	97.316	*9786*	*9786*	0	64.868
700_700_0.15_0.85	9121	8985.51	65.90	507.656	9177	9003.15	138.46	659.194	*9229*	9155.79	18.61	445.194	*9229*	9187.55	20.70	486.304	*9229*	**9229**	0	96.472
800_800_0.10_0.75	9890	9656.38	51.42	567.090	*9932*	9823.17	113.20	607.506	*9932*	9685.79	72.06	868.227	*9932*	9930.56	14.33	214.286	*9932*	**9932**	0	21.032
800_800_0.15_0.85	8961	8774.18	59.78	161.688	8907	8732.94	160.07	590.883	8961	8909.50	10.91	27.170	*9101*	8936.12	39.55	321.859	*9101*	**9101**	0	129.395
900_900_0.10_0.75	9526	9462.86	37.83	670.990	*9745*	9639.60	51.13	598.520	*9745*	9660.12	36.68	341.110	*9745*	9729.51	30.06	368.807	*9745*	**9745**	0	45.950
900_900_0.15_0.85	8718	8492.88	62.31	702.655	8916	8617.20	210.54	665.798	8916	8916	0	116.694	*8990*	8918.96	14.50	672.574	*8990*	**8990**	0	237.865
1000_1000_0.10_0.75	9348	9250.80	53.65	542.187	9509	9273.64	82.57	802.652	9544	9255.73	142.33	876.669	9544	9431.47	60.84	510.660	**9551**	**9551**	0	142.712
1000_1000_0.15_0.85	8330	8037.92	71.87	932.614	8134	7872.84	95.76	97.909	8379	8206.49	68.52	632.334	8474	8376.20	27.12	500.435	**8538**	**8497.39**	28.46	505.954
585_600_0.10_0.75	10300	10161.45	72.81	98.186	*10393*	10191.01	102.35	729.422	*10393*	10366.15	29.83	499.311	*10393*	*10393*	0	89.785	*10393*	*10393*	0	73.093
585_600_0.15_0.85	9031	8944.22	61.72	616.631	*9256*	*9256*	0	103.637	*9256*	*9256*	0	264.876	*9256*	*9256*	0	84.359	*9256*	*9256*	0	99.163
685_700_0.10_0.75	10070	9953.55	49.02	430.180	*10121*	9909	30.82	123.012	*10121*	9979.70	86.13	540.289	*10121*	10114.96	31.87	230.918	*10121*	**10121**	0	9.229
685_700_0.15_0.85	9102	8860.79	106.42	159.976	*9176*	8936.47	135.64	645.153	*9176*	9139.18	52.80	461.051	*9176*	*9176*	0	140.151	*9176*	*9176*	0	96.859
785_800_0.10_0.75	9123	8885.09	54.14	316.494	*9384*	9163.90	70.91	339.415	*9384*	9236.10	95.56	576.738	*9384*	**9384**	0	136.173	*9384*	9382.68	9.24	210.315
785_800_0.15_0.85	8556	8482.33	51.45	604.625	8572	8322.17	57.53	665.514	8663	8558.51	79.51	586.047	*8746*	8643.93	47.92	467.334	*8746*	**8684.58**	36.41	720.765
885_900_0.10_0.75	9137	9079.09	46.70	590.376	9232	9121.24	48.92	455.104	9232	9106.31	62.28	452.360	*9318*	9236.16	21.32	281.632	*9318*	**9318**	0	81.932
885_900_0.15_0.85	8217	7881.44	65.84	140.935	8277	7900.57	131.65	296.061	*8425*	8268	104.34	484.859	*8425*	8311.68	46.80	625.829	*8425*	**8411.72**	9.88	573.526
985_1000_0.10_0.75	9067	8994.48	44.99	313.094	9113	8938.38	66.64	967.315	9047	8917.48	126.37	89.760	9193	9105.84	74.76	319.356	**9234**	**9193.15**	13.26	855.645
985_1000_0.15_0.85	8453	8425.27	48.74	503.976	8172	7958.24	121.56	350.640	8528	8233.05	119.98	283.901	8528	8488.13	33.47	450.711	**8612**	**8578.20**	32.47	628.435
#Avg	9196.67	9041.40	62.54	482.096	9280.33	9105.91	85.91	499.248	9310.10	9202.09	57.96	473.031	9352.30	9303.10	23.95	379.479	**9362.93**	**9351.59**	7.22	280.940

Table 3
Summarized comparisons of the MSBTS algorithm against each reference algorithm over the two sets of benchmark instances.

Algorithm pair	Instance set	Indicator	#Wins	#Ties	#Losses	*p-value*
MSBTS vs. DHJaya [17]	Set I (30)	f_{best}	16	14	0	4.82e-4
		f_{avg}	23	6	1	1.37e-4
	Set II (30)	f_{best}	30	0	0	1.82e-06
		f_{avg}	30	0	0	1.86e-09
MSBTS vs. HBPSO/TS [21]	Set I (30)	f_{best}	2	28	0	1.80e-1
		f_{avg}	11	12	7	1.33e-1
	Set II (30)	f_{best}	20	10	0	5.96e-5
		f_{avg}	29	1	0	2.56e-6
MSBTS vs. I2PLS [9]	Set I (30)	f_{best}	0	30	0	NA
		f_{avg}	19	5	6	2.64e-2
	Set II (30)	f_{best}	15	15	0	8.83e-5
		f_{avg}	29	1	0	2.56e-6
MSBTS vs. KBTS [22]	Set I (30)	f_{best}	0	30	0	NA
		f_{avg}	6	11	13	9.10e-2
	Set II (30)	f_{best}	7	23	0	1.80e-2
		f_{avg}	24	5	1	1.57e-5

the Wilcoxon signed-rank test in the last row.

From Table 4, we observe that the proposed MSBTS algorithm is very competitive compared to the reference algorithms in terms of T_{best} and T_{avg}. In particular, MSBTS attains the smallest T_{best} values for 22 instances (out of 30) against 0 for DHJaya, HBPSO/TS, I2PLS and 8 for KBTS. Also, MSBTS has a better average performance according to the #Avg values in the last row. The *p-values* (< 0.05) from the Wilcoxon signed-rank test clearly indicate that differences between MSBTS and the compared algorithms are statistically significant.

To further illustrate the computational efficiency of MSBTS compared to the reference algorithms, we plot the points (T_i, ρ_i) based on two SUKP instances of Set II and show the time-to-target plots in Fig. 5. The X-axis in each sub-figure indicates the time to achieve the target value, and the Y-axis is the cumulative probability of reaching the given target value. We observe that the cumulative probability of each algorithm increases with the run-time. However, MSBTS (also KBTS) attains a high probability (over 90%) in a very short computation time (less than 20 seconds) on both instances, while the other algorithms perform poorly. Regarding MSBTS and KBTS, in order to attain a probability of 99.5% of reaching the target value, MSBTS requires about 12 seconds on both instances, while KBTS consumes around 42 seconds and 26 seconds. Note that DHJaya failed to obtain the probability of 99.5% within the time limit of 1000s on both instances. This experiment demonstrates the computational efficiency of the proposed MSBTS algorithm.

Table 4
Time-to-target analysis on the SUKP instances of Set II.

Instance/Algorithm	Target	DHJaya		HBPSO/TS		I2PLS		KBTS		MSBTS	
		$T_{best}(s)$	$T_{avg}(s)$	$T_{best}(s)$	$T_{avg}(s)$	$T_{best}(s)$	$T_{avg}(s)$	$T_{best}(s)$	$T_{avg}(s)$	$T_{best}(s)$	$T_{avg}(s)$
600_585_0.10_0.75	9500	100.079	523.459	3.470	9.353	3.741	11.927	**0.213**	**0.618**	0.654	1.297
600_585_0.15_0.85	9100	65.826	566.872	67.883	382.176	6.599	59.784	3.486	12.187	**1.244**	**9.123**
700_685_0.10_0.75	9700	270.577	561.072	11.334	133.119	11.657	66.970	1.216	7.799	**0.858**	**5.310**
700_685_0.15_0.85	9100	106.614	427.274	123.888	526.406	9.663	178.041	1.647	42.561	**1.370**	**23.398**
800_785_0.10_0.75	9500	160.885	650.605	18.534	132.560	25.917	241.929	**1.272**	15.965	1.325	**7.600**
800_785_0.15_0.85	8700	151.590	516.174	68.445	323.062	15.246	102.963	5.448	55.774	**2.492**	**7.798**
900_885_0.10_0.75	9400	313.696	560.054	37.409	271.706	13.254	295.578	**1.530**	28.776	3.346	**8.834**
900_885_0.15_0.85	8400	221.799	400.128	499.176	652.865	13.318	459.241	2.592	60.691	**2.139**	**9.243**
1000_985_0.10_0.75	9000	291.897	421.090	9.114	97.051	13.008	150.602	1.061	21.855	**0.639**	**16.614**
1000_985_0.15_0.85	8300	293.618	574.331	678.089	820.440	530.745	530.745	**6.685**	116.893	10.870	**25.255**
600_600_0.10_0.75	10500	67.558	369.584	16.691	51.810	5.938	31.448	2.589	58.678	**1.041**	**5.271**
600_600_0.15_0.75	8800	68.179	560.449	6.067	131.515	5.670	40.425	1.170	**5.538**	**0.697**	5.450
700_700_0.10_0.75	9500	654.112	743.459	9.297	163.108	9.090	99.874	1.769	12.083	**0.721**	**4.817**
700_700_0.15_0.85	9100	105.922	521.651	111.166	690.543	23.306	265.033	4.807	29.800	**2.098**	**19.571**
800_800_0.10_0.75	9800	573.460	576.004	180.088	549.453	866.553	866.553	9.431	213.756	**6.618**	**21.098**
800_800_0.15_0.85	8800	162.727	575.454	114.424	508.682	15.821	131.655	**1.385**	27.487	2.459	**8.224**
900_900_0.10_0.75	9500	220.422	603.266	33.222	261.629	11.589	46.073	**1.275**	8.965	2.809	**6.297**
900_900_0.15_0.85	8600	235.578	459.369	50.142	554.410	12.601	84.906	1.033	10.397	**0.912**	**8.208**
1000_1000_0.10_0.75	9300	327.772	784.859	76.998	560.236	30.069	412.291	**2.125**	149.036	18.468	**43.389**
1000_1000_0.15_0.85	8000	294.562	530.699	76.860	548.614	25.684	225.339	2.132	24.634	**1.218**	**8.561**
585_600_0.10_0.75	10000	64.865	245.444	15.053	83.161	6.935	17.042	1.614	5.787	**0.746**	**2.510**
585_600_0.15_0.85	9000	65.337	528.534	8.954	63.449	7.319	96.275	1.033	21.288	**1.005**	**21.108**
685_700_0.10_0.75	10000	333.101	472.050	137.383	171.016	108.642	484.414	13.512	235.948	**2.818**	**5.230**
685_700_0.15_0.85	9000	154.648	514.173	189.391	531.964	19.709	299.990	3.127	45.425	**1.370**	**29.075**
785_800_0.10_0.75	8900	155.496	484.831	9.029	96.090	11.278	104.360	**0.756**	7.486	0.765	**2.847**
785_800_0.15_0.85	8500	150.938	607.258	679.254	679.254	27.872	358.703	10.115	155.450	**2.401**	**22.656**
885_900_0.10_0.75	9100	222.106	619.250	30.648	425.582	36.186	415.666	6.096	73.726	**4.159**	**17.378**
885_900_0.15_0.85	8000	346.018	631.780	228.520	564.195	28.099	209.716	1.941	17.235	**1.746**	**7.186**
985_1000_0.10_0.75	8900	300.232	540.491	278.500	651.225	36.254	428.971	12.316	113.230	**9.698**	**48.404**
985_1000_0.15_0.85	8100	281.460	437.529	109.148	276.985	47.763	287.634	1.088	12.129	**0.997**	**10.208**
#Avg	9073	225.369	533.573	129.272	363.722	65.984	233.472	3.482	53.040	**2.923**	**13.732**
#Best	-	0	0	0	0	0	0	8	2	**22**	**28**
#*p-value*	-	1.86e-09	1.86e-09	1.86e-09	1.86e-09	1.86e-09	1.86e-09	2.48e-2	9.31e-09	-	-

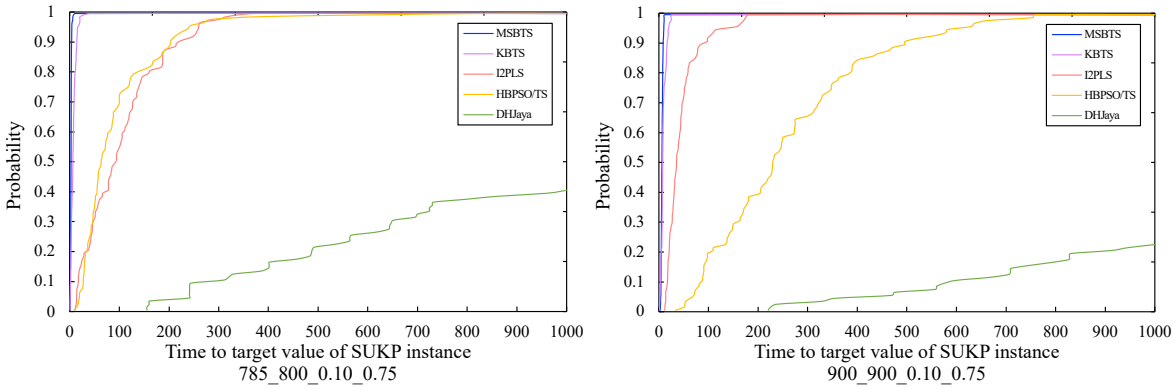

Fig. 5. Cumulative probability distributions for the time to reach a target value.

4 Analysis

In this section, we perform additional experiments to investigate the influences of the main ingredients of the MSBTS algorithm. Specifically, we study the effect of the parameter γ_v of the hash functions (Section 4.1), the error rates of hash functions (Section 4.2) and the benefit of the solution-based tabu search strategy (Section 4.3).

4.1 Sensitivity analysis of hash functions

Hash functions are the key ingredients of the MSBTS algorithm. Now, we analyze the influence of the parameter γ_v ($v = 1, 2, 3$) involved in the hash functions (see Section 2.4.2) on the performance of the MSBTS algorithm. As indicated in [31], the proper settings of γ_v should satisfy two conditions: (1) the hash values of each candidate solution should be no more than the allowed maximum integer to avoid overflow; (2) the distribution of hash values of different candidate solutions should be wide enough to reduce possible collisions. We have carried out preliminary experiments for γ_v used in the hash functions. Experimental results show that a large γ_v value (> 2.8) will lead to integer overflow for instances with more than 985 items or elements. On the other hand, a small γ_v value (< 1.0) will lead to the same values of $\lfloor \mathcal{W}_i^v \rfloor$ ($\mathcal{W}_i^v = i^{\gamma_v}$) for adjacent items, increasing the probability of collisions. For example, assuming γ_v=0.9, the $\lfloor \mathcal{W}_i^v \rfloor$ values of the adjacent items 501 and 502 are both 269 ($\mathcal{W}_{501} = 501^{0.9} = 269.06$, $\mathcal{W}_{502} = 502^{0.9} = 269.55$). Given two neighboring solutions S_1 and $S_2 = S_1 \oplus swap(500, 501)$ where the *swap* operator was defined in Section 2.4.1, they will get the same hash value. As we focus on the ranges (1.0, 2.8) to analyze the influence of the parameter γ_v.

For this purpose, we tested different groups of parameters (γ_1, γ_2, γ_3) (see Table 5) on 10 representative SUKP instances, i.e., 785_800_0.15_0.85, 800_785_0.10_0.75, 800_785_0.15_0.85, 885_900_0.15_0.85, 900_885_0.15_0.85, 985_1000_0.10_0.75, 985_1000_0.15_0.85, 1000_985_0.10_0.75, 1000_985_0.15_0.85, 1000_1000_0.15_0.85. These 10 instances are denoted by G_1 to G_{10} in Table 5, respectively. For the experiment, we performed 30 independent runs for each setting of parameters on each instance with the cut-off time of 1000 seconds, and recorded the average objective values (f_{avg}). In fact, we do not provide the best object values (f_{best}) here, since most of the f_{best} values obtained with different group of parameters (γ_1, γ_2, γ_3) are exactly the same.

Table 5 displays the comparative results of this experiment, where the first row shows the label of each tested instance and the first column indicates the

setting of the parameters (γ_1, γ_2, γ_3). The f_{avg} values of each group of γ_v are shown in rows 2 to 21, respectively. In addition, the last row #std gives the standard deviation of each column and the last column #Avg presents that the average values of each row.

From Table 5, we observe that the parameter γ_v is not sensitive for our algorithm. First, the results obtained from different groups of parameters are very similar in terms of #Avg values. Specially, there are 12 out of 20 groups of parameters that obtained the same f_{avg} value on instance G_4. Second, the small #std values of each column indicate that the standard deviations of the results shown in the columns are relatively low. The *p-value* of 0.633 (> 0.05) from the Friedman statistical test again confirms that there are no significant differences among the tested results. This analysis indicated that any γ_v value in the interval (1.0, 2.8) is suitable for the proposed algorithm.

Table 5
Influence of the hash functions on the average performance of MSBTS algorithm.

(γ_1, γ_2, γ_3)/ Instance	G_1	G_2	G_3	G_4	G_5	G_6	G_7	G_8	G_9	G_{10}	#Avg
(1.1,1.3,1.5)	8665.77	9930.33	9004	8408.87	8578.23	9190.07	*8579.50*	**9647.67**	**8453.80**	8491.90	8895.01
(1.1,1.5,1.9)	8687.90	*9937*	8985.50	8411	8577.10	9191.87	8575.17	9627.50	8453.27	8490.07	8893.64
(1.2,1.4,1.8)	8687.90	*9937*	8983.67	8412.20	**8579.27**	9190.73	8583.83	9638.50	8453.60	8487.93	8895.46
(1.2,1.6,2.0)	*8693.43*	*9937*	8992.83	*8413.80*	8576.07	9192.53	*8579.50*	9631.60	8453.20	**8500.37**	**8897.03**
(1.3,1.5,1.7)	8671.30	*9937*	9000.17	8411.67	8581.43	**9192.93**	8577.33	9636.10	8453.43	8490.77	8895.21
(1.3,1.7,2.1)	8690.67	*9937*	8985.50	*8413.80*	8566.33	9189.03	8577.33	9631.60	8453.13	8491.70	8893.61
(1.4,1.6,2.0)	8687.90	9933.67	**8994.67**	8411.47	8582.53	9191.80	8573	9626.13	8453.27	8486.93	8894.14
(1.5,1.7,1.9)	8679.60	*9937*	8989.17	8412.20	8568.43	9189.73	8573	9631	8453.33	8496.10	8892.96
(1.5,1.9,2.3)	8685.13	9933.67	8983.67	8412.20	8571.67	9191.80	*8579.50*	9636.80	8453.40	8492.47	8894.03
(1.6,1.8,2.2)	8679.60	*9937*	8989.17	8412.20	8568.43	9189.73	8573	9631	8453.33	8496.10	8892.96
(1.7,1.9,2.1)	8685.13	9933.67	8985.50	8412.20	8573.90	9190.50	8573	9637.50	8453.40	8492.47	8893.73
(1.7,2.1,2.5)	8682.37	9933.67	8981.83	8412.20	8573.90	9191.57	8575.17	9637.50	8453.40	8492.47	8893.41
(1.8,2.0,2.4)	8685.13	9933.67	8983.67	8412.20	8571.67	9191.80	*8579.50*	9636.80	8453.40	8492.47	8894.03
(1.9,2.1,2.3)	8682.37	9933.67	8981.83	8412.20	8573.90	9191.57	8577.33	9637.20	8453.40	8492.47	8893.59
(1.9,2.3,2.7)	8682.37	9933.67	8981.83	8412.20	8573.90	9191.57	8577.33	9637.50	8453.40	8492.47	8893.62
(2.0,2.2,2.6)	8685.13	9933.67	8985.50	8412.20	8573.90	9190.50	8573	9637.50	8453.40	8492.47	8893.73
(1.1,1.2,2.7)	*8693.43*	*9937*	8983.67	8410.60	8568.40	9191.20	8573	9636.67	8453.40	8494.33	8894.17
(1.1,1.8,2.7)	8679.60	9933.67	8985.50	8412.73	8571.73	9191.57	8573	9643	8453.40	8490.33	8893.45
(1.1,2.0,2.7)	8676.83	9933.67	8981.83	8412.20	8573.90	9191.57	8575.17	9637.20	8453.40	8492.47	8892.82
(1.1,2.5,2.7)	8676.83	9933.67	8981.83	8412.20	8571.73	9191.57	8575.17	9637.20	8453.33	8492.47	8892.60
#std	6.93	1.96	6.30	1.04	4.35	0.99	3.10	4.94	0.14	2.88	-

4.2 Error rates of hash functions

An error occurs when an unvisited solution is wrongly forbidden by the hash functions and the associated hash vectors. To calculate the error rates of hash functions, we ran our SBTS procedure for 10^4 iterations on two SUKP instances: 1000_1000_0.10_ 0.75, 1000_1000_0.15_0.85. During the search, each

encountered solution is recorded in a pool POP. We use a counter c_1 to count the number of solutions forbidden (classified as tabu) by the hash functions. Another counter c_2 (error counter) will be incremented by 1 if the solution is not included in POP. Then the error rate is obtained by c_2/c_1. We perform additional experiments to investigate two factors that affect the error rates of the hash functions: 1) the length L of the hash functions, and 2) the number of the hash vectors.

The role of the length L is to ensure that the hash vectors are long enough to be able to record the sampled solutions. A proper setting of L should not only avoid memory overflow, but also keep the error rates at a low level. The results of preliminary experiments indicate that a large L value ($> 10^8$) leads to memory overflow. Thus we carried out an experiment to check the error rates of the hash vectors with L ranging from 10^5 to 10^8. The error rate plots are shown in Fig. 6, where the iterations of the SBTS procedure and the corresponding error rates are displayed on the X-axis and the Y-axis, respectively.

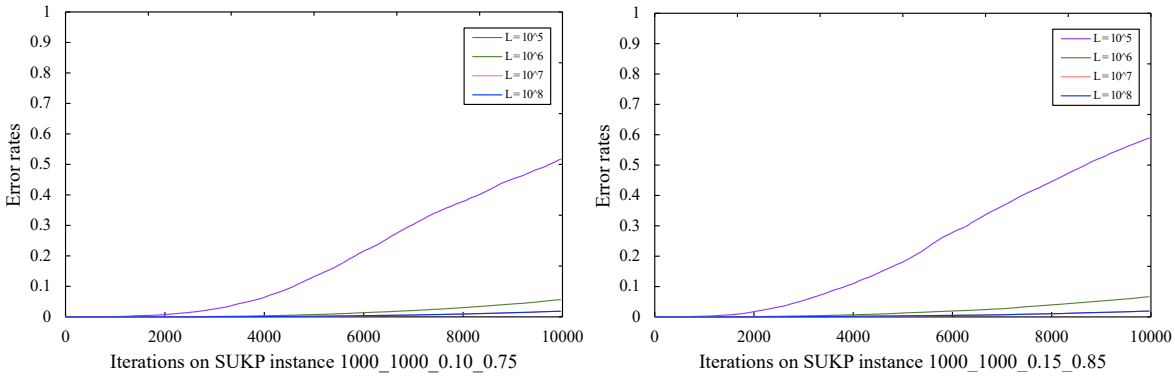

Fig. 6. Impact of the length L of hash vectors on the error rate of the solution-based tabu search procedure.

Fig. 6 shows that our algorithm can keep the error rate at a low level (< 0.07) with L ranging from 10^6 to 10^8. In particular, when the values of L are 10^7 and 10^8, the corresponding curves almost overlap and stay under 0.02. The error rates increase dramatically (more than 0.5) as the number of iterations increases for $L \leq 10^5$. Considering that the time complexity of evaluating a neighboring solution is $O(1)$, a large L value will not significantly affect the computation time. Thus any L value in the interval $[10^6, 10^8]$ is suitable for the proposed algorithm ($L = 10^8$ in the MSBTS algorithm).

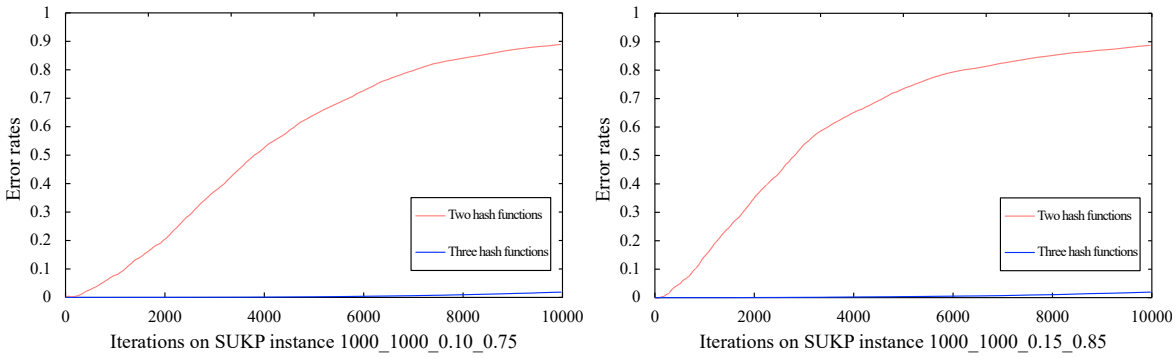

Fig. 7. Impact of the number of hash vectors on the error rate of the solution-based tabu search procedure.

The role of the hash vectors is to record the solutions encountered during the search, and the number of the hash vectors can significantly affect the error rates. We performed another experiment to analyze the error rates when using two or three hash vectors. As the error rate plots in Fig. 7 show, the SBTS procedure has an error rate of nearly 0.9 with two hash vectors over 10^4 iterations. The error rate with one hash vector will be naturally higher than that with two hash vectors for the same number of iterations. On the contrary, the error rate remains very low (< 0.02) with three hash vectors over 10^4 iterations. So three hash vectors can effectively identify the previously encountered solutions, which justifies the use of three hash vectors in MSBTS.

4.3 Analysis of solution-based tabu search

The solution-based tabu search strategy is the most innovative component of the MSBTS algorithm. To understand its influence on the algorithm, we created a MSBTS variant; named MABTS where the solution-based tabu search procedure is replaced by an attributed-based tabu search procedure. For this experiment, we employed the attributed-based tabu search method introduced in [22], which is one of the best SUKP algorithm. Thus, except the tabu search procedure, MABTS shares the other components of MSBTS.

Considering that our algorithm mainly shows its superiority on the large instances, we carried out this experiment based on Set II, where each instance was independently solved by each algorithm 30 times, each run being limited to 1000 seconds.

The experimental results are reported in Table 6. The first column shows the names of the instances. The results of the two compared algorithms are respectively presented in columns 2 to 7, including the best objective value (f_{best}), the average objective value (f_{avg}), the standard deviation over 30 runs

(*std*). To facilitate the comparison, we also provide the similar #Avg, #Best and *p-values* as described in Section 3.4.

Table 6 shows that MSBTS significantly outperforms MABTS, achieving better f_{best} values (marked in bold) for 17 out of the 30 instances and equal results for the remaining 13 instances. When comparing the f_{avg} values, MSBTS again dominates MABTS for all the instances. Moreover, the *std* values of MSBTS are very small, indicating that MSBTS is highly robust. Furthermore, the small *p-values* (< 0.05) show that there is a significant difference between MSBTS and MABTS. This experiment confirms that the solution-based tabu search strategy constitutes one key ingredient of our algorithm.

5 Conclusions

The Set-Union Knapsack Problem attracts more and more attention in recent years due to its theoretical and practical interest. Inspired by the fact that the solution-based tabu search has been successfully applied to solve several difficult binary optimization problems, we devised the first multistart solution-based tabu search algorithm for solving SUKP. The proposed MSBTS algorithm uses its solution-based tabu search procedure to find high-quality local optima and the multistart mechanism to overcome deep local optima traps. MSBTS has several desirable features such as simple design and implementation as well as absence of parameters.

We performed extensive experimental assessments of the proposed algorithm on two sets of 60 benchmark instances. The comparisons with the state-of-the-art algorithms demonstrated the high competitiveness of our algorithm in terms of solution quality, computational efficiency and robustness. In particular, we demonstrated the interest of the MSBTS algorithm to deal with large instances and reported new lower bounds for 7 large and difficult instances (with 585 to 1000 items and elements).

This work thus provides a useful tool for solving the general Set-Union Knapsack Problem. Moreover, since a number of real-world applications can be conveniently formulated by SUKP, the proposed algorithm can be hopefully applied to these practical problems. The availability of the code of the MSBTS algorithm will facilitate such applications.

Finally, this work provides another supporting evidence for using the solution-based tabu search strategy to solve binary problems. As such it would be interesting to verify the effectiveness of this approach on additional problems including those related to knapsack.

Table 6
Comparison between MSBTS and MABTS on the instances of Set II.

Instance/Setting	MSBTS			MABTS		
	f_{best}	f_{avg}	std	f_{best}	f_{avg}	std
600_585_0.10_0.75	*9914*	**9914**	0	*9914*	9801.57	72.65
600_585_0.15_0.85	*9357*	**9357**	0	*9357*	9329.40	23.76
700_685_0.10_0.75	**9881**	**9881**	0	9841	9814.37	34.83
700_685_0.15_0.85	**9163**	**9163**	0	9135	9126.67	14.16
800_785_0.10_0.75	**9937**	**9937**	0	9811	9679.73	61.37
800_785_0.15_0.85	*9024*	**8992.83**	27.25	*9024*	8892.53	51.21
900_885_0.10_0.75	**9725**	**9725**	0	9611	9503.63	53.57
900_885_0.15_0.85	**8620**	**8576.07**	27.14	8499	8459.87	26.51
1000_985_0.10_0.75	**9689**	**9631.60**	28.92	9580	9411.37	58.09
1000_985_0.15_0.85	**8455**	**8453.20**	0.60	8448	8359.30	106.74
600_600_0.10_0.75	*10524*	**10524**	0	*10524*	10519.67	3.54
600_600_0.15_0.75	*9062*	**9062**	0	*9062*	9058.20	11.40
700_700_0.10_0.75	*9786*	**9786**	0	*9786*	9770.20	37.93
700_700_0.15_0.85	**9229**	**9229**	0	9177	9145.20	30.65
800_800_0.10_0.75	*9932*	**9932**	0	*9932*	9734.87	64.12
800_800_0.15_0.85	**9101**	**9101**	0	8956	8907.10	14.88
900_900_0.10_0.75	**9745**	**9745**	0	9660	9629.20	36.02
900_900_0.15_0.85	**8990**	**8990**	0	8916	8911.03	17.49
1000_1000_0.10_0.75	**9551**	**9551**	0	9357	9269.87	92.10
1000_1000_0.15_0.85	**8538**	**8500.37**	28.65	8381	8282.20	73.08
585_600_0.10_0.75	*10393*	**10393**	0	*10393*	10325.43	34.75
585_600_0.15_0.85	*9256*	**9256**	0	*9256*	9256	0
685_700_0.10_0.75	*10121*	**10121**	0	*10121*	9944.10	59.12
685_700_0.15_0.85	*9176*	**9176**	0	*9176*	9144.97	31.29
785_800_0.10_0.75	*9384*	**9384**	0	*9384*	9229.37	93.68
785_800_0.15_0.85	**8746**	**8693.43**	40.00	8663	8526.57	59.71
885_900_0.10_0.75	**9318**	**9318**	0	9232	9158.57	40.38
885_900_0.15_0.85	*8425*	**8413.80**	7.33	*8425*	8276.07	42.39
985_1000_0.10_0.75	**9234**	**9192.53**	14.12	9193	9030.77	54.53
985_1000_0.15_0.85	**8612**	**8579.50**	32.50	8461	8384.43	75.03
#Avg	**9362.93**	**9352.61**	6.88	9309.17	9229.41	45.83
#Best	**30**	**30**	-	13	0	-
p-value	2.93e-4	2.563e-06	-	-	-	-

Declaration of competing interest

The authors declare that they have no known competing interests that could have appeared to influence the work reported in this paper.

Acknowledgments

We are grateful to the reviewers for their useful comments and suggestions which helped us to significantly improve the paper. We would like to thank Dr. Congcong Wu, Dr. Geng Lin, Dr. Xiangjing Lai and their co-authors for sharing the codes of their algorithms ([17], [21], [32]). Support from the China Scholarship Council (Grant 201706290016) for the first author is also acknowledged.

References

[1] O. Goldschmidt, D. Nehme, G. Yu, Note: On the set-union knapsack problem, Nav. Res. Log. 41 (6) (1994) 833–842.

[2] H. Kellerer, U. Pferschy, D. Pisinger, Knapsack problems, Springer, 2004.

[3] B. Schneier, Applied cryptography - protocols, algorithms, and source code in C, 2nd Edition, John Wiley & Sons, 1996.

[4] S. B. Navathe, S. Ceri, G. Wiederhold, J. Dou, Vertical partitioning algorithms for database design, ACM Trans. Database Syst. 9 (4) (1984) 680–710.

[5] M. Tu, L. Xiao, System resilience enhancement through modularization for large scale cyber systems, in: 2016 IEEE/CIC International Conference on Communications in China (ICCC Workshops), IEEE, 2016, pp. 1–6.

[6] W. D. Lister, R. G. Laycock, A. M. Day, A key-pose caching system for rendering an animated crowd in real-time, Comput. Graph. Forum 29 (8) (2010) 2304–2312.

[7] A. Arulselvan, A note on the set union knapsack problem, Discret. Appl. Math. 169 (2014) 214–218.

[8] R. Taylor, Approximations of the densest k-subhypergraph and set union knapsack problems, arXiv preprint :1610.04935 (2016).

[9] Z. Wei, J. Hao, Iterated two-phase local search for the set-union knapsack problem, Future Gener. Comput. Syst. 101 (2019) 1005–1017.

[10] Y. He, H. Xie, T. Wong, X. Wang, A novel binary artificial bee colony algorithm for the set-union knapsack problem, Future Gener. Comput. Syst. 78 (2018) 77–86.

[11] F. B. Özsoydan, A. Baykasoglu, A swarm intelligence-based algorithm for the set-union knapsack problem, Future Gener. Comput. Syst. 93 (2019) 560–569.

[12] A. Baykasoğlu, F. B. Ozsoydan, M. E. Senol, Weighted superposition attraction algorithm for binary optimization problems, Oper. Res. (2018) 1–27.

[13] F. B. Özsoydan, Artificial search agents with cognitive intelligence for binary optimization problems, Comput. Ind. Eng. 136 (2019) 18–30.

[14] Y. He, X. Wang, Group theory-based optimization algorithm for solving knapsack problems, Knowl. Based Syst. (2018) 104445.

[15] Y. Feng, H. An, X. Gao, The importance of transfer function in solving set-union knapsack problem based on discrete moth search algorithm, Mathematics 7 (1) (2019) 17.

[16] Y. Feng, J. Yi, G. Wang, Enhanced moth search algorithm for the set-union knapsack problems, IEEE Access 7 (2019) 173774–173785.

[17] C. Wu, Y. He, Solving the set-union knapsack problem by a novel hybrid jaya algorithm, Soft Comput. 24 (3) (2020) 1883–1902.

[18] X. Liu, Y. He, Estimation of distribution algorithm based on lévy flight for solving the set-union knapsack problem, IEEE Access 7 (2019) 132217–132227.

[19] I. Gölcük, F. B. Ozsoydan, Evolutionary and adaptive inheritance enhanced grey wolf optimization algorithm for binary domains, Knowl. Based Syst. 194 (2020) 105586.

[20] H. H. Hoos, T. Stützle, Stochastic Local Search: Foundations & Applications, Elsevier / Morgan Kaufmann, 2004.

[21] G. Lin, J. Guan, Z. Li, H. Feng, A hybrid binary particle swarm optimization with tabu search for the set-union knapsack problem, Expert Syst. Appl. 135 (2019) 201–211.

[22] Z. Wei, J. Hao, Kernel based tabu search for the set-union knapsack problem, Expert Syst. Appl. 165 (2020) 113802.

[23] F. Glover, M. Laguna, Tabu search, Springer Science+Business Media New York, 1997.

[24] Y. Lu, B. Cao, C. Rego, F. Glover, A tabu search based clustering algorithm and its parallel implementation on spark, Appl. Soft Comput. 63 (2018) 97 – 109.

[25] I. Polak, M. Boryczka, Tabu search in revealing the internal state of rc4+ cipher, Appl. Soft Comput. 77 (2019) 509 – 519.

[26] Q. Zhou, J. Hao, Z. Sun, Q. Wu, Memetic search for composing medical crews with equity and efficiency, Appl. Soft Comput. 94 (2020) 106440.

[27] W. B. Carlton, J. W. Barnes, A note on hashing functions and tabu search algorithms, Eur. J. Oper. Res. 95 (1) (1996) 237–239.

[28] D. L. Woodruff, E. Zemel, Hashing vectors for tabu search, Annals OR 41 (2) (1993) 123–137.

[29] X. Lai, J. Hao, F. W. Glover, Z. Lü, A two-phase tabu-evolutionary algorithm for the 0-1 multidimensional knapsack problem, Inf. Sci. 436-437 (2018) 282–301.

[30] X. Lai, J. Hao, D. Yue, Two-stage solution-based tabu search for the multidemand multidimensional knapsack problem, Eur. J. Oper. Res. 274 (1) (2019) 35–48.

[31] Y. Wang, Q. Wu, F. W. Glover, Effective metaheuristic algorithms for the minimum differential dispersion problem, Eur. J. Oper. Res. 258 (3) (2017) 829–843.

[32] X. Lai, D. Yue, J. Hao, F. W. Glover, Solution-based tabu search for the maximum min-sum dispersion problem, Inf. Sci. 441 (2018) 79–94.

[33] J. Chang, L. Wang, J. Hao, Y. Wang, Parallel iterative solution-based tabu search for the obnoxious p-median problem, Comput. Oper. Res. 127 (2020) 105155.

[34] E. D. N. Ruiz, X. Yang, Improved tabu search and simulated annealing methods for nonlinear data assimilation, Appl. Soft Comput. 83 (2019).

[35] M. Shahmanzari, D. Aksen, A multi-start granular skewed variable neighborhood tabu search for the roaming salesman problem, Appl. Soft Comput. (2020) 107024.

[36] Y. Chen, J. Hao, An iterated "hyperplane exploration" approach for the quadratic knapsack problem, Comput. Oper. Res. 77 (2017) 226–239.

[37] R. M. Aiex, M. G. C. Resende, C. C. Ribeiro, TTT plots: a perl program to create time-to-target plots, Optim. Lett. 1 (4) (2007) 355–366.

[38] C. C. Ribeiro, I. Rosseti, R. V. Campos, Exploiting run time distributions to compare sequential and parallel stochastic local search algorithms, J. Global Optimization 54 (2) (2012) 405–429.