Iterated two-phase local search for the Set-Union Knapsack Problem

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

[a] *LERIA, Université d'Angers, 2 Boulevard Lavoisier, 49045 Angers, France*
[b] *Institut Universitaire de France, 1 Rue Descartes, 75231 Paris, France*

Future Generation Computer Systems
https://doi.org/10.1016/j.future.2019.07.062

Abstract

Many practical decision-making problems involve selecting a subset of objects from a set of candidate objects such that the selected objects optimize a given objective while satisfying some constraints. Knapsack problems such as the Set-union Knapsack Problem (SUKP) are general models that allow such decision-making problems to be conveniently formulated. Given a set of weighted elements and a set of items with profits where each item is composed of a subset of elements, the SUKP aims to pack a subset of items in a capacity-constrained knapsack in a way that the total profit of the selected items is maximized while their weights do not exceed the knapsack capacity. In this work, we present an effective iterated two-phase local search algorithm for this NP-hard problem. The proposed algorithm iterates through two complementary search phases: a local optima exploration phase to discover local optimal solutions, and a local optima escaping phase to drive the search to unexplored regions. We show the competitiveness of the algorithm compared to the state-of-the-art methods in the literature. Specifically, the algorithm discovers 18 improved best results (new lower bounds) for the 30 benchmark instances and matches the best-known results for the 12 remaining instances. We also report the first computational results with the general CPLEX solver, including 6 proven optimal solutions. Finally, we investigate the impacts of the key ingredients of the algorithm on its performance.

Keywords: Knapsack problems; Computational methods; Heuristics and metaheuristics; Combinatorial optimization.

* Corresponding author.
Email address: `jin-kao.hao@univ-angers.fr` (Jin-Kao Hao).

1 Introduction

Knapsack problems are very general and useful models able to formulate numerous real-world problems in a variety of fields. For instance, suppose that a firm has a fixed global budget envelope for project investment as well as a number of candidate projects. Suppose also that each candidate project requires a budget and its implementation implies a gain. One important decision problem is then to select a subset of projects from the candidate set such that the total gain of the retained projects is maximized and the total budget allocated to the retained projects is no more than the available budget envelope. This practical problem as well as many other similar problems can conveniently be formulated with the following general 0/1 knapsack problem (KP) [17]. Given a knapsack with a weight capacity and a set of items where each item has a weight and a profit. The KP involves selecting a subset of items in a way that the total profit of the selected items is maximized, while the weight sum of the selected items does not exceed the knapsack capacity. As indicated in [17], the KP can be used to model many real-world decision-making problems such as selection of investments and portfolios, generating keys for cryptosystems, and finding the least wasteful way to cut raw materials.

The Set-union Knapsack Problem (SUKP) studied in this work is a generalized knapsack problem that can formulate additional applications. Let $U = \{1, \ldots, n\}$ be a set of n elements with weights $w_j > 0$ $(j = 1, \ldots, n)$. Let $V = \{1, \ldots, m\}$ be a set of m items where each item i $(i = 1, \ldots, m)$ corresponds to a subset of elements $U_i \subset U$ determined by a relation matrix and has a profit $p_i > 0$. For an arbitrary non-empty item set $S \subset V$, the total profit of S is defined as $f(S) = \sum_{i \in S} p_i$, and the weight of S is given by $W(S) = \sum_{j \in \cup_{i \in S} U_i} w_j$. Let $C > 0$ be the capacity of a given knapsack, the SUKP involves finding a subset of items $S^* \subset V$ such that the profit $f(S^*)$ is maximized and the weight $W(S^*)$ does not surpass the knapsack capacity C. Formally, the SUKP can be stated as follows [13].

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

$$\text{s.t.} \quad W(S) = \sum_{j \in \cup_{i \in S} U_i} w_j \leq C, \ S \subset V \tag{2}$$

It is worth noting that for a given subset S of items, the weight w_j of an element j is counted only once in $W(S)$ even if the element belongs to more than one selected items.

One notices that the conventional knapsack problem is a special case of the SUKP. Indeed, the SUKP reduces to the KP when we set $m = n$ and $V = U$.

The SUKP also generalizes the NP-hard densest k-subhypergraph problem (DkSH) that aims to determine a set of k nodes of a hypergraph to maximize the number of hyperedges of the subhypergraph induced by the set of the selected nodes [4]. In fact, the SUKP reduces to the DkSH when we consider the elements and items as the nodes and hyperedges of a hypergraph respectively, with unit weights and unit profits as well as a capacity of k. As indicated in [1,13,14], the SUKP is an useful model for a number of relevant applications, such as financial decision making, flexible manufacturing, building public key prototype, database partitioning etc. However, as a generalization of the NP-hard KP and DkSH problems, the SUKP is computationally challenging.

Given its theoretical and practical significance, the SUKP has received more and more attention. As the review in Section 2 shows, various search methods have been proposed in the literature, including exact, approximation and metaheuristic algorithms. In particular, recent studies focused on metaheuristic algorithms which aim to find satisfactory solutions as fast as possible, without optimality guarantee of the attained solutions. These algorithms are especially useful to handle large and difficult problem instances when they cannot be solved by exact approaches. We observe that the state-of-the-art algorithms such as [9,14,22] all adopted swam optimization metaheuristics. However, given that these methods are initially designed for solving continuous problems, the swam optimization based algorithms for the SUKP simulate discrete optimization via continuous search operators, instead of exploring the discrete space directly. As such, applying swam optimization to the SUKP requires various adaptations to cope with the binary feature of the SUKP. In particular, these algorithms must adopt an empirical transfer function to map the continuous space to the discrete space and maintain both continuous and discrete solutions during the search. Moreover, as indicated in [14], these approaches inevitably generate infeasible solutions, and therefore need a repairing procedure to handle these infeasible solutions.

In this work, we show for the first time that stochastic local search, which directly operates in the binary search space, can be a highly effective approach for solving the SUKP. The work is motivated by two considerations. First, stochastic local search has been quite successful in solving numerous challenging combinatorial problems [15], including several knapsack problems such as multidimensional knapsack problem [11,18,25], multidemand multidimensional knapsack problem [5,19], multiple-choice multidimensional knapsack problem [6,16], quadratic knapsack problem [8,27], quadratic multiple knapsack problem [7,23] and generalized quadratic knapsack problem [2]. Second, given that the SUKP is basically a constrained subset selection problem with binary variables, it is natural to investigate solution methods that explore the binary search space and focus on feasible solutions. Indeed, as we show in this work, our discrete optimization approach based on stochastic local search is quite valuable for the SUKP.

The contributions of this work are summarized as follows.

(1) From a perspective of algorithm design, the proposed iterated two-phase local search algorithm relies on two innovative and complementary search components specially designed for the SUKP. The intensification-oriented component (first phase) employs a combined neighborhood search strategy to discover local optimal solutions. The diversification-oriented component (second phase) helps the search process to explore unvisited regions. The combination of these two complementary search phases enables the algorithm to perform an effective examination of the search space.
(2) From a perspective of computational performance, we show the competitiveness of the proposed algorithm compared to the state-of-the-art algorithms on the set of 30 benchmark instances commonly used in the literature. In particular, we report improved best results for 18 large instances and equal best results for the 12 remaining instances. The improved best results (new lower bounds) are useful for future studies on the problem, e.g., they can serve as references for evaluating existing and new SUKP algorithms.
(3) Third, we investigate for the first time the interest of the general mixed integer programming solver CPLEX for solving the SUKP. We show that while CPLEX (version 12.8) can find the optimal solutions for the 6 small benchmark instances (with 85 to 100 items and elements) based on a simple 0/1 linear programming model, it fails to exactly solve the other 24 instances. These outcomes provide strong motivations for developing effective approximate algorithms to handle problem instances that cannot be solved exactly.
(4) This work demonstrates that the discrete optimization approach based on stochastic local search is quite valuable and effective for solving the SUKP. This work invites thus more investigations in this direction, in addition to the swarm optimization based approaches.

The remaining part of this paper is organized as follows. In Section 2, we provide a review of the related work on solution methods for the SUKP. In Section 3, we present the general framework of the proposed algorithm as well as its composing ingredients. Computational results and comparisons with the best-performing algorithms and CPLEX are reported in Section 4. In Section 5, we analyze the parameters and components of the algorithm and show their effects on its performance. In the last section, we summarize the present work and discuss future research directions.

2 Related Work

This section is dedicated to a review of existing solution methods for the SUKP.

First, in 1994, Goldschmidt et al. devised an exact algorithm based on the general dynamic programming method [13] and presented sufficient conditions for it to run in polynomial time. In 2014, Arulselvan studied a greedy algorithm that is based on a previous approximation algorithm for the related budgeted maximum coverage problem [1]. The algorithm provides a $(1 - e^{-\frac{1}{d}})$ approximation for the SUKP with the additional restriction that the number of items in which an element is present is bounded by a constant d. In 2016, Taylor designed an approximation algorithm using results of the related densest k-subhypergraph problem [24]. The proposed algorithm is shown to achieves, for any given $\epsilon > 0$, an approximation ratio of at most $O(n^{\alpha_m+\epsilon})$ for $\alpha_m = \frac{2}{3}[m - 1 - \frac{2m-2}{m2+m-1}]$, where the subsets have at most m elements. Focusing on theoretical aspects of the SUKP, these studies do not show computational results.

Second, in addition to these theoretical studies, metaheuristic algorithms based on swarm optimization were recently studied to find sub-optimal solutions for the SUKP [9,14,22]. In 2018, He et al. proposed the first binary artificial bee colony algorithm (BABC) for solving the SUKP [14]. Since this approach inevitably generates infeasible solutions, a greedy repairing and optimization procedure (named S-GROA) is proposed to handle infeasible solutions. To assess their algorithm, He et al. generated a set of 30 random instances (more information on these instances can be found in Section 4.1) and presented large scale experiments on these instances. Comparisons with three other population-based algorithms (genetic algorithm, continuous artificial bee colony algorithm, differential evolution strategies) showed the competitiveness of the BABC algorithm. In 2019, Ozsoydan and Baykasoglu presented a binary particle swarm optimization algorithm (gPSO) and reported improved best results on the set of 30 benchmark instances [22]. Also in 2019, Feng et al. investigated several versions of discrete moth search (MS) and reported computational results on 15 out of the 30 benchmark instances with some updated best results [9]. It is worth mentionning that like BABC, both gPSO and MS generate infeasible solutions during the search process and use the S-GROA procedure of [14] to recover solution feasibility.

In this work, we are interested in practical solving of the SUKP with heuristics and investigate the first binary optimization approach based on stochastic local search. To highlight the technical contributions of this work, we provide in Table 1 the main characteristics of the proposed algorithm with respect to the reviewed studies. From Table 1, we observe that compared to the existing methods, our approach distinguishes itself by its adopted search framework, the search strategy and the explored search space. As we show in Section 4, this approach is indeed very effective for solving the SUKP and competes favorably with the state-of-the-art algorithms presented in [9,14,22] on the set of 30 benchmark instances commonly tested in the literature.

Table 1
Summary of the main features and technical contributions of the proposed algorithm with respect to the most related studies.

Algorithm	Type of approach	Framework	Search strategy	Search space	Comments	Computational results
[13] (1994)	Exact	Dynamic programming	Implicit exhaustive enumeration	Binary space	Theoretical guarantee of optimality, exponential time complexity	No
[1] (2014)	Approximation	Greedy	Progressive construction guided by a greedy function	Binary space	Theoretical guarantee of an approximation ratio	No
[24] (2016)	Approximation	Greedy	Progressive construction guided by a greedy function	Binary space	Theoretical guarantee of an approximation ratio	No
[14] (2018)	Heuristic	Artificial bee colony optimization	Solution combination; repair of infeasible solutions, mapping between continuous solution and discrete solution	Continuous and binary spaces including both infeasible and feasible binary solutions	Suboptimal solutions, polynomial time complexity	Yes
[9] (2019)	Heuristic	Moth search	Solution combination, repair of infeasible solutions, mapping between continuous solution and discrete solution	Continuous and binary spaces including both infeasible and feasible binary solutions	Suboptimal solutions, polynomial time complexity	Yes
[22] (2019)	Heuristic	Particle swarm optimization, genetic algorithm	Solution combination, repair of infeasible solutions, mapping between continuous solution and discrete solution	Continuous and binary spaces including both infeasible and feasible solutions	Suboptimal solutions, polynomial time complexity	Yes
This work	Heuristic	Stochastic local search	Neighborhoods based iterative improvement, perturbation based diversification, focus on feasible solution	Binary space including only feasible solutions	Suboptimal solutions, polynomial time complexity	Yes

3 Iterated two-phase local search for the SUKP

This section is dedicated to the presentation of the proposed iterated two-phase local search algorithm (I2PLS) for the SUKP. We first show its general scheme, and then explain the composing ingredients.

3.1 General Algorithm

As shown in Algorithm 1, I2PLS is composed of two complementary search phases: a local optima exploration phase (Explore) to find new local optimal solutions of increasing quality and a local optima escaping phase (Escape) to displace the search to unexplored regions.

Algorithm 1 Iterated two-phase local search for the SUKP

```
1: **Input**: Instance $I$, cut-off time $t_{max}$, neighborhoods $N_1 - N_3$, exploration depth
   $\lambda_{max}$, sampling probability $\rho$, tabu search depth $\omega_{max}$, perturbation strength $\eta$.
2: **Output**: The best solution found $S^*$.
3: /* Generate an initial solution $S_0$ in a greedy way, §3.3                          */
   $S_0 \leftarrow Greedy\_Initial\_Solution(I)$
4: $S^* \leftarrow S_0$                    /* Record the overall best solution $S^*$ found so far */
5: **while** $Time \leq t_{max}$ **do**
6:    /* Local optima exploration phase using VND and TS, §3.4                      */
      $S_b \leftarrow$VND-TS$(S_0, N_1 - N_3, \lambda_{max}, \rho, \omega_{max})$
7:    **if** $f(S_b) > f(S^*)$ **then**
8:       $S^* \leftarrow S_b$                  /* Update the best solution $S^*$ found so far */
9:    **end if**
10:   /* Local optima escaping phase using frequency-based perturbation, §3.5 */
      $S_0 \leftarrow$Frequency_Based_Local_Optima_Escaping$(S_b, \eta)$
11: **end while**
12: **return** $S^*$
```

The algorithm starts from a feasible initial solution (line 3, Alg. 1) that is obtained with a greedy construction procedure (Section 3.3). Then it enters the 'while' loop to iterate the 'Explore' phase and the 'Escape' phase (lines 5-11, Alg. 1) to seek solutions of improving quality. At each iteration, the 'Explore' phase (line 6, Alg. 1) first performs a variable neighborhood descent (VND) search to locate a new local optimal solution within two neighborhoods N_1 and N_2 and then runs a tabu search (TS) to explore additional local optima with a different neighborhood N_3 (Section 3.4). When the 'Explore' phase is exhausted, I2PLS switches to the 'Escape' phase (line 10, Alg. 1), which uses a frequency-based perturbation to displace the search to an unexplored region (Section 3.5). These two phases are iterated until a stopping condition (in our case, a given time limit t_{max}) is reached. During the search process, the best solution found is recorded in S^* (lines 7-8, Alg. 1) and returned as the final output of the algorithm at the end of the algorithm.

One notices that the general scheme of the I2PLS algorithm for the SUKP shares ideas of breakout local search [3], three-phase local search [10] and iterated local search [20]. Meanwhile, to ensure its effectiveness for solving the SUKP, the proposed algorithm integrates dedicated search components tailored for the considered problem, which are described below.

3.2 Solution Representation, Search Space, and Evaluation Function

Given a SUKP instance composed of m items $V = \{1, \ldots, m\}$, n elements $U = \{1, \ldots, n\}$ and knapsack capacity C. The search space Ω includes all non-empty subsets of items such that the capacity constraint is satisfied, i.e., $\Omega = \{S \subset V : S \neq \emptyset, \sum\limits_{j \in \cup_{i \in S} U_i} w_j \leq C\}$.

For any candidate solution S of Ω, its quality is assessed by the objective value $f(S) = \sum\limits_{i \in S} p_i$ that corresponds to the total profit of the selected items.

Notice that a candidate solution S of Ω can be represented by $S =< A, \bar{A} >$ where A is the set of selected items and $\bar{A}$ are the non-selected items. Equivalently S can also be coded by a binary vector of length m where each binary variable corresponds to an item and its value indicates whether the item is selected or not selected.

The goal of our I2PLS algorithm is to find a solution $S \in \Omega$ with the objective value $f(S)$ as large as possible.

3.3 Initialization

The I2PLS algorithm starts its search with an initial solution, which is generated by a simple greedy procedure in three steps. First, we calculate the total weight w_i of each item i in $O(mn)$. Second, based on the given profit p_i of each item, we obtain the *profit ratio* r_i of each item by $r_i = p_i/w_i$ and sort all items in the descending order according to r_i in $O(log(m))$. Third, we add one by one the items to S by following this order until the capacity of the knapsack is reached in $O(m)$. The time complexity of the initialization procedure is thus $O(mn)$.

3.4 Local Optima Exploration Phase

Algorithm 2 Local Optima Exploration Phase - VND-TS

```
1: Input: Starting solution $S$, neighborhoods $N_1 - N_3$, exploration depth $\lambda_{max}$,
   sampling probability $\rho$, tabu search depth $\omega_{max}$,
2: Output: The best solution $S_b$ found by VND-TS.
3: $S_b \leftarrow S$            /*$S_b$ records the best solution found so far during VND-TS */
4: $\lambda \leftarrow 0$            /*$\lambda$ counts the number of consecutive non-improving rounds*/
5: while $\lambda < \lambda_{max}$ do
6:     /* Attain a new local optimum $S$ by VND with $N_1$ and $N_2$, see Alg. 3     */
       $S \leftarrow$VND$(S, N_1, N_2, \rho)$
7:     /* Explore nearby optima around the new $S$ by TS with $N_3$, see Alg. 5     */
       $(S_c, S) \leftarrow$TS$(S, N_3, \omega_{max})$ /*$S_c$ is the best solution found so far during TS */
8:     if $f(S_c) > f(S_b)$ then
9:         $S_b \leftarrow S_c$                    /* Update the best solution $S_b$ found so far */
10:        $\lambda \leftarrow 0$
11:    else
12:        $\lambda \leftarrow \lambda + 1$
13:    end if
14: end while
15: return $S_b$
```

From an initial solution, the 'Explore' phase (see Algorithm 2) aims to find new local optimal solutions of increasing quality. This is achieved by a combined strategy mixing a variable neighborhood descent (VND) procedure (line 6, Alg. 2, see Section 3.4.1) and a tabu search (TS) procedure (line 7, Alg. 2, see Section 3.4.3). For each VND-TS run (each 'while' iteration), the VND procedure exploits, with the best-improvement strategy, two neighborhoods N_1 and N_2 to locate a local optimal solution. Then from this solution, the TS procedure is triggered to examine additional local optimal solutions with another neighborhood N_3. At the end of TS, its best solution (S_c) is used to update the recorded best solution (S_b) found during the current VND-TS run, while its last solution (S) is used as the new starting point of the next iteration of the 'Explore' phase. The 'Explore' phase terminates when the best solution (S_b) found during this run cannot be updated during λ_{max} consecutive iterations (λ_{max} is a parameter called *exploration depth*).

3.4.1 Variable Neighborhood Descent Search

Following the general variable neighborhood descent search [21], the VND procedure (Algorithm 3) relies on two neighborhoods (N_1 and N_2, see Sections 3.4.2) to explore the search space. Specifically, VND examines the neighborhood N_1 at first and iteratively identifies a best-improving neighbor solution in N_1 to replace the current solution. When a local optimal solution is reached within N_1, VND switches to the neighborhood N_2. As we explain in Section 3.4.2, given the large size of N_2, VND only examines a subset N_2^- which is composed of $\rho \times |N_2|$ randomly solutions of N_2 (ρ is a parameter

Algorithm 3 Variable Neighborhood Descent - VND

```
 1: Input: Input solution $S$, neighborhoods $N_1$ and $N_2$, sampling probability $\rho$.
 2: Output: The best solution $S_b$ found during the VND search.
 3: $S_b \leftarrow S$                    /*$S_b$ record the best solution found so far*/
 4: $Improve \leftarrow True$
 5: while $Improve$ do
 6:   $S \leftarrow argmax\{f(S') : S' \in N_1(S)\}$
 7:   if $f(S) > f(S_b)$ then
 8:     $S_b \leftarrow S$                /*Update the best solution found so far*/
 9:     $Improve = True$
10:   else
11:     $N_2^- \leftarrow Sampling(N_2, S, \rho)$
12:     $S \leftarrow argmax\{f(S') : S' \in N_2^-(S)\}$
13:     if $f(S) > f(S_b)$ then
14:       $S_b \leftarrow S$              /*Update the best solution found so far*/
15:       $Improve \leftarrow True$
16:     else
17:       $Improve = False$
18:     end if
19:   end if
20: end while
21: return $S_b$
```

Algorithm 4 Sampling Procedure

```
1: Input: Input solution $S$, neighborhood $N_2$, sampling probability $\rho$.
2: Output: Set $N_2^-$ of sampled solutions from $N_2(S)$
3: $N_2^- \leftarrow \emptyset$
4: for each $S' \in N_2(S)$ do
5:   if $random() < \rho$ then
6:     $N_2^- \leftarrow N_2^- \cup \{S'\}$
7:   end if
8: end for
9: return $N_2^-$
```

called sampling probability and Algorithm 4 shows the sampling procedure where *random*() is a random real number in [0,1]). If an improving neighbor solution is detected in N_2^-, VND switches back to N_1. VND terminates when no improving solution can be found within both neighborhoods. In Section 5.2, we study the influence of this sampling strategy.

3.4.2 Move Operators, Neighborhoods and VND Exploration

To explore candidate solutions of the search space, the I2PLS algorithm employs the general *swap* operator to transform solutions. Specifically, let $S =< A, \bar{A} >$ be a given solution with A and $\bar{A}$ being the set of selected and non-selected items. Let $swap(q, p)$ denote the operation that deletes q items from A and adds p other items from $\bar{A}$ into A. By limiting q and p to specific values, we introduce two particular $swap(q,p)$ operators.

The first operator $swap_1(q,p)$ ($q \in \{0,1\},\ p = 1$) includes two customary operations as described in the literature [19,26,28]: the *Add* operator and the *Exchange* operator. Basically, $swap_1(q,p)$ either adds an item from $\bar{A}$ into A or exchanges one item in A with another item in $\bar{A}$ while keeping the capacity constraint satisfied.

The second operator $swap_2(q,p)$ ($3 \le q+p \le 4$) covers three different cases: delete two items from A and add one item from $\bar{A}$ into A; delete one item from A and add two items from $\bar{A}$ into A; exchanges two items of A against two items of $\bar{A}$. These three operations are subject to the capacity constraint.

On the basis of these two swap operators, we define the neighborhoods N_1^w and N_2^w induced by $swap_1$ and $swap_2$ as follows.

$$N_1^w(S) = \{S' : S' = S \oplus swap_1(q,p), q \in \{0,1\},\ p = 1, \sum_{j \in \cup_{i \in S'} U_i} w_j \le C\} \quad (3)$$

$$N_2^w(S) = \{S' : S' = S \oplus swap_2(q,p), 3 \le q+p \le 4, \sum_{j \in \cup_{i \in S'} U_i} w_j \le C\} \quad (4)$$

where $S' = S \oplus swap_k(q,p)$ ($k = 1,2$) is the neighbor solution of the incumbent solution S obtained by applying $swap_1(q,p)$ or $swap_2(q,p)$ to S.

N_1^w and N_2^w are bounded in size by $O(|A| \times |\bar{A}|)$ and $O(\binom{2}{|A|} \times \binom{2}{|\bar{A}|})$ respectively.

Given the large sizes of these neighborhoods, it is obvious that exploring all the neighbor solutions at each iteration will be very time consuming. To cope with this problem, we adopt the idea of a filtering strategy that excludes the non-promising neighbor solutions from consideration [19]. Specifically, a neighbor solution S' qualifies as promising if $f(S') > f(S_b)$ holds, where S_b is the best solution found so far in Algorithm 3. Using this filtering strategy, we define the following reduced neighborhoods N_1 and N_2.

$$N_1(S) = \{S' \in N_1^w(S) : f(S') > f(S_b)\} \quad (5)$$

$$N_2(S) = \{S' \in N_2^w(S) : f(S') > f(S_b)\} \quad (6)$$

As explained in Section 3.4.1 and Algorithm 3, the VND procedure successively examines solutions of these two neighborhoods N_1 and N_2. Notice that $swap_2$ leads generally to a very large number of neighbor solutions such that even the reduced neighborhood N_2 can still be too large to be explored efficiently. For this reason, the VND procedure explores a sampled portion of N_2 at each iteration, according to the sampling procedure shown in Algorithm 4.

3.4.3 Tabu Search

Algorithm 5 Tabu Search - TS

```
 1: Input: Input solution $S$, Neighborhood $N_3$, tabu search depth $\omega_{max}$
 2: Output: The best solution $S_b$ found during tabu search, the last solution $S$ of
    tabu search.
 3: $S_b \leftarrow S$                      /*$S_b$ records the best solution found so far*/
 4: $\omega \leftarrow 0$        /*$\omega$ counts the number of consecutive non-improving iterations */
 5: while $\omega < \omega_{max}$ do
 6:   $S \leftarrow argmax\{f(S') : S' \in N_3(S)$ and $S'$ is not forbidden by the $tabu\_list\}$
 7:   if $f(S) > f(S_b)$ then
 8:     $S_b \leftarrow S$                  /* Update the best solution $S_b$ found so far */
 9:     $\omega \leftarrow 0$
10:   else
11:     $\omega \leftarrow \omega + 1$
12:   end if
13:   Update the $tabu\_list$
14: end while
15: return $(S_b, S)$
```

To discover still better solutions when the VND search terminates, we trigger the tabu search (TS) procedure (Algorithm 5) that is adapted from the general tabu search metaheuristic [12]. To explore candidate solutions, TS relies on the $swap_3(q,p)$ $(1 \leq p+q \leq 2)$ operator, which extends $swap_1$ used in VND by including the case $q = 1, p = 0$, which corresponds to the drop operation (i.e., deleting an item from A without adding any new item). One notices that $swap_3(1,0)$ always leads to a neighbor solution of worse quality, which can be usefully selected for search diversification. We use N_3 to denote the neighborhood induced by $swap_3$.

$$N_3(S) = \{S' : S' = S \oplus swap_3(q,p), 1 \leq p+q \leq 2, \sum_{j \in \cup_{i \in S'} U_i} w_j \leq C\} \qquad (7)$$

As shown Algorithm 5, the TS procedure iteratively makes transitions from the incumbent solution S to a selected neighbor solution S' in N_3. At each iteration, TS selects the best neighbor solution S' in N_3 (or one of the best ones if there are multiple best solutions) that is not forbidden by the so-called tabu list (*tabu_list*) (line 6, Alg. 5, see below). Notice that if no improving solution exists in $N_3(S)$, the selected neighbor solution S' is necessarily a worsening or equal-quality solution relative to S. It is this feature that allows TS to go beyond local optimal traps. To prevent the search from revisiting previously encountered solutions, the tabu list is used to record the items involved in the swap operation. And each item i of the tabu list is then forbidden to take part in any swap operation during the next T_i consecutive iterations where T_i is called the tabu tenure of item i and is empirically fixed as follows.

$$T_i = \begin{cases} 0.4 \times |A|, & \text{if } i \in A; \\ 0.2 \times |\bar{A}| \times (100/m), & \text{if } i \in \bar{A}. \end{cases} \tag{8}$$

TS terminates when its best solution cannot be further improved during ω_{max} consecutive iterations (ω_{max} is a parameter called the tabu search depth).

3.5 Frequency-Based Local Optima Escaping Phase

The 'Explore' phase aims to diversify the search by exploring new search regions. For this purpose, the algorithm keeps track of the frequencies that each item has been displaced and uses the frequency information to modify (perturb) the incumbent solution. Particularly, we adopt an integer vector F of length m whose elements are initialized to zero. Each time an item i is displaced by a swap operation, F_i is increased by one. Thus, items with a low frequency are those that are not frequently moved during the 'Explore' phase. Then when the 'Explore' phase terminates and before the next round of the 'Explore' phase starts, we modify the best solution $S_b =< A_b, \bar{A}_b >$ as follows. We delete the top $\eta \times |A_b|$ least frequently moved items from A_b (η is a parameter called *perturbation strength* and adds to A_b randomly select items from $\bar{A}_b$ until the knapsack capacity is reached. This perturbed solution serves as the new starting solution S_0 of the next iteration of the algorithm (see line 10, Alg. 1). In Section 5.3, we study the usefulness of this perturbation strategy.

4 Experimental Results and Comparisons

This section presents a performance assessment of the I2PLS algorithm. We show computational results on the 30 benchmark instances commonly used in the literature, in comparison with three state-of-the-art algorithms for SUKP. We also present the first results from the CPLEX solver.

4.1 Benchmark Instances

We use the 30 benchmark instances provided in [14], which were also tested in 2 other recent studies [9,22]. These instances are divided into three sets according to the relationship between the number of items and elements ranging from 85 to 500, where each instance has a different density α of elements in an item and a different ratio β of the knapsack capacity to the total weight of all elements. Let R be a $m \times n$ binary relation matrix between m items and n elements where $R_{ij} = 1$ indicates the presence of element j in item i, w_j be the weight of element j, and C the knapsack capacity. Then $m\_n\_\alpha\_\beta$ designates an instance with m items and n elements, density of α and ratio of β, where $\alpha = (\sum_{i=1}^{m} \sum_{j=1}^{n} R_{ij})/(mn)$ and $\beta = C/\sum_{j=1}^{n} w_j$. The characteristics of the three sets of instances are shown in Tables 3 to 5. These 30 instances are denoted by F1−F10 ($m > n$), S1−S10 ($m = n$) and T1−T10 ($m < n$) in Fig. 1 to 2, respectively.

4.2 Experimental Setting and reference algorithms

The proposed algorithm was implemented in C++ and compiled using the g++ compiler with the -O3 option. The experiments were carried on an Intel Xeon E5-2670 processor with 2.5 GHz and 2 GB RAM under the Linux operating system.

Table 2
Settings of parameters.

Parameters	Sect.	Description	Value
λ_{max}	2	Exploration depth	2
ρ	3.4.1	Sampling probability for VND	5
ω_{max}	3.4.3	Tabu search depth	100
η	3.5	Perturbation strength in escaping phase	0.5

Table 2 shows the setting of parameters used in our algorithm, whose values were discussed in Section 5.1. Given the stochastic nature of the algorithm, we ran 100 times (like in [14,22]) with different random seeds to solve each instance, with a cut-off time of 500 seconds per run.

For the comparative studies, we use as reference algorithms the following three very recent algorithms: BABC (binary artificial bee colony algorithm) (2018), which is the best performing among five population-based algorithms tested in [14], gPSO (binary particle swarm optimization algorithm) (2019) [22] and MS (discrete moth search algorithm) (2019) [9]. Among these reference algorithms, we obtained the code of BABC. So for BABC, we report both the results listed in [14] as well as the results obtained by running the BABC code on our computer under the same time limit of 500 seconds. For gPSO and MS, we cite the results reported in the corresponding papers. The results of these reference algorithms have been obtained on computing platforms with the following features: an Intel Core i5-3337u processor with 1.8 GHz and 4 GB RAM for BABC, an Intel Core i7-4790K 4.0 GHz processor with 32 GB RAM for gPSO, and an Intel Core i7-7500 processor with 2.90 GHz and 8.00 GB RAM for MS.

Additionally, we notice that until now, no result has been reported by using the general integer linear programming (ILP) approach to solve the SUKP. Therefore, we include in our experimental study the results achieved by the ILP CPLEX solver (version 12.8) under a time limit of 2 hours based on the 0/1 linear programming model presented in the Appendix.

4.3 Computational Results and Comparisons

The computational results [1] of I2PLS on the three sets of benchmark instances are reported in Tables 3-5, together with the results of the reference algorithms (BABC [14], gPSO [22], MSO4 [9]) where BABC* corresponds to the results by running the BABC code as explained in Section 4.2 and MSO4 is the best MS version among all twelve MS algorithms studied in [9]. The first column of each table gives the name of each instance. Column 2 (Best_Known) indicates the best known value reported in the literature and compiled from [9,14,22]. The best lower bound (LB) and upper bound (UB) achieved by the CPLEX solver are given in columns 3 and 4. Column 5 lists respectively the four performance indicators: best objective value (f_{best}), average objective value over 100 runs (f_{avg}), standard deviations over 100 runs (std), and average run times t_{avg} in seconds to reach the best objective value. Columns 6 to 9 present the computational statistics of the compared algorithms. The best values of f_{best} and f_{avg} among the results of the compared algorithms are highlighted in bold and the equal values are indicated in italic. Entries with "-" mean that the results are not available.

Given the fact that the compared algorithms were run on different computing platforms and they report solutions of various quality, it is not meaningful to compare the computation times. Therefore, the comparisons are mainly based on the quality, while run times (when they are available) are included only for indicative purposes.

Finally, Table 6 provides a summary of the compared algorithms on all 30 benchmark instances where rows $\#Better$, $\#Equal$ and $\#Worse$ indicate the number of instances for which each algorithm obtains a better, equal or worse f_{best} value compared to the best-known values in the literature (Best_Known). Moreover, to further analyze the performance of our I2PLS algorithm, we use the non-parametric Wilcoxon signed-rank test to check the statistical significance of the compared results between I2PLS and each reference algorithm in terms of f_{best} values. The outcomes of the Wilcoxon tests are shown in the last row of Table 6 where a *p-value* smaller than 0.05

[1] Our solution certificates are available at: `http://www.info.univ-angers.fr/pub/hao/SUKP_I2PLS.html`. The code of I2PLS will also be made available.

implies a significant performance difference between I2PLS and its competitor.

Table 3
Computational results and comparison of the proposed I2PLS algorithm with the reference algorithms on the first set of instances ($m > n$).

Instance	Best_Known	LB	UB	Results	BABC	BABC*	gPSO	MSO4	I2PLS
100_85_0.10_0.75*	*13283*	*13283*	*13283*	f_{best}	13251	*13283*	*13283*	*13283*	*13283*
				f_{avg}	13208.5	*13283*	13050.53	13062	*13283*
				std	92.63	0	37.41	-	0
				t_{avg}	0.210	51.102	-	1.398	3.094
100_85_0.15_0.85*	12274	*12479*	*12479*	f_{best}	12238	*12479*	12274	-	*12479*
				f_{avg}	12155	**12479**	12084.82	-	12335.13
				std	53.29	0	95.38	-	98.78
				t_{avg}	0.223	24.032	-	-	103.757
200_185_0.10_0.75	*13521*	11585	27055.82	f_{best}	13241	13402	13405	*13521*	*13521*
				f_{avg}	13064.4	13260.16	13286.56	13193	**13521**
				std	99.57	38.98	93.18	-	0
				t_{avg}	1.562	253.693	-	7.901	71.984
200_185_0.15_0.85	14044	11017	29625.82	f_{best}	13829	*14215*	14044	-	*14215*
				f_{avg}	13359.2	14026.18	13492.60	-	**14031.28**
				std	234.99	151.55	328.72	-	131.46
				t_{avg}	1.729	241.932	-	-	180.809
300_285_0.10_0.75	11335	9028	43937.51	f_{best}	10428	10572	11335	11127	**11563**
				f_{avg}	9994.76	10466.45	10669.51	10302	**11562.02**
				std	154.03	61.94	227.85	-	3.94
				t_{avg}	5.281	315.240	-	24.912	181.248
300_285_0.15_0.85	12245	6889	53164.23	f_{best}	12012	12245	12245	-	**12607**
				f_{avg}	10902.9	12019.28	11607.10	-	**12364.55**
				std	449.45	85.76	477.80	-	83.03
				t_{avg}	5.673	226.818	-	-	240.333
400_385_0.10_0.75	*11484*	8993	66798.30	f_{best}	10766	11021	*11484*	11435	*11484*
				f_{avg}	10065.2	10608.91	10915.87	10411	**11484**
				std	241.45	138.07	367.75	-	0
				t_{avg}	12.976	293.560	-	56.838	31.801
400_385_0.15_0.85	10710	5179	77480.39	f_{best}	9649	9649	10710	-	**11209**
				f_{avg}	9135.98	9503.65	9864.55	-	**11157.26**
				std	151.90	94.69	315.38	-	87.29
				t_{avg}	13.359	270.813	-	-	141.525
500_485_0.10_0.75	11722	7202	86166.50	f_{best}	10784	10927	11722	11031	**11771**
				f_{avg}	10452.2	10628.31	11184.51	10716	**11729.76**
				std	114.35	70.31	322.98	-	6.59
				t_{avg}	25.372	486.210	-	124.378	349.545
500_485_0.15_0.85	10022	4762	97218.01	f_{best}	9090	9306	10022	-	**10238**
				f_{avg}	8857.89	9014.01	9299.56	-	**10133.94**
				std	94.55	64.06	277.62	-	94.72
				t_{avg}	26.874	482.740	-	-	369.375

From Tables 3 to 5, we observe that our I2PLS algorithm performs extremely well compared to the state-of-the-art results on the set of 30

Table 4
Computational results and comparison of the proposed I2PLS algorithm with the reference algorithms on the second set of instances ($m = n$).

Instance	Best_Known	LB	UB	Results	BABC	BABC*	gPSO	MSO4	I2PLS
100_100_0.10_0.75*	*14044*	*14044*	*14044*	f_{best}	13860	*14044*	*14044*	*14044*	*14044*
				f_{avg}	13734.9	14040.87	13854.71	13649	**14044**
				std	70.76	11.51	96.23	-	0
				t_{avg}	0.213	169.848	-	1.646	38.245
100_100_0.15_0.85*	*13508*	*13508*	*13508*	f_{best}	*13508*	*13508*	*13508*	-	*13508*
				f_{avg}	13352.4	**13508**	13347.58	-	13451.50
				std	155.14	0	194.34	-	126.49
				t_{avg}	0.244	6.795	-	-	70.587
200_200_0.10_0.75	*12522*	11187	29394.32	f_{best}	11846	12350	*12522*	12350	*12522*
				f_{avg}	11194.3	11953.11	11898.73	11508	**12522**
				std	249.58	97.57	391.83	-	0
				t_{avg}	1.633	183.130	-	8.112	54.780
200_200_0.15_0.85	*12317*	9258	30610.99	f_{best}	11521	11929	*12317*	-	*12317*
				f_{avg}	10945	11695.21	11584.64	-	**12280.07**
				std	255.14	78.33	275.32	-	57.77
				t_{avg}	1.819	147.930	-	-	238.348
300_300_0.10_0.75	12736	11007	45191.75	f_{best}	12186	12304	12695	12598	**12817**
				f_{avg}	11945.8	12202.80	12411.27	11541	**12817**
				std	127.80	67.81	225.80	-	0
				t_{avg}	5.315	202.515	-	28.612	66.403
300_300_0.15_0.85	11425	7590	51891.53	f_{best}	10382	10857	11425	-	**11585**
				f_{avg}	9859.69	10383.64	10568.41	-	**11512.18**
				std	177.02	75.79	327.48	-	73.15
				t_{avg}	6.019	113.380	-	-	220.100
400_400_0.10_0.75	11531	7910	68137.98	f_{best}	10626	10869	11531	10727	**11665**
				f_{avg}	10101.1	10591.65	10958.96	10343	**11665**
				std	196.99	105.83	274.90	-	0
				t_{avg}	12.805	298.970	-	58.433	18.733
400_400_0.15_0.85	10927	4964	77719.78	f_{best}	9541	10048	10927	-	**11325**
				f_{avg}	9032.95	9602.13	9845.17	-	**11325**
				std	194.18	142.77	358.91	-	0
				t_{avg}	12.953	386.555	-	-	76.000
500_500_0.10_0.75	10888	7500	85184.48	f_{best}	10755	10755	10888	10355	**11249**
				f_{avg}	10328.5	10522.56	10681.46	9919	**11243.40**
				std	94.62	70.17	125.36	-	27.43
				t_{avg}	27.735	194.490	-	121.622	134.186
500_500_0.15_0.85	10194	3948	101964.36	f_{best}	9318	9601	10194	-	**10381**
				f_{avg}	9180.74	9334.52	9703.62	-	**10293.89**
				std	84.91	40.59	252.84	-	85.53
				t_{avg}	27.813	135.130	-	-	237.894

benchmark instances. In particular, I2PLS improves on the best-known results of the literature for 18 out of 30 instances, while matching the best-known results for the remaining 12 instances. Notice that among these 12 instances, 6 instances with 85 and 100 items are solved to optimality by

Table 5
Computational results and comparison of the proposed I2PLS algorithm with the reference algorithms on the third set of instances ($m < n$).

Instance	Best_Known	LB	UB	Results	BABC	BABC*	gPSO	MSO4	I2PLS
85_100_0.10_0.75*	*12045*	*12045*	*12045*	f_{best}	11664	*12045*	*12045*	11735	*12045*
				f_{avg}	11182.7	11995.12	11486.95	11287	**12045**
				std	183.57	53.15	137.52	-	0
				t_{avg}	0.188	206.570	-	1.354	2.798
85_100_0.15_0.85*	*12369*	*12369*	*12369*	f_{best}	*12369*	*12369*	*12369*	-	*12369*
				f_{avg}	12081.6	**12369**	11994.36	-	12315.53
				std	193.79	0	436.81	-	62.60
				t_{avg}	0.217	0.531	-	-	17.47
185_200_0.10_0.75	*13696*	12264	25702.48	f_{best}	13047	13647	*13696*	13647	*13696*
				f_{avg}	12522.8	13179.14	13204.26	13000	**13695.60**
				std	201.35	100.78	366.56	-	3.68
				t_{avg}	1.502	202.560	-	7.642	124.136
185_200_0.15_0.85	11298	8608	26289.16	f_{best}	10602	10926	*11298*	-	*11298*
				f_{avg}	10150.6	10749.46	10801.41	-	**11276.17**
				std	152.91	97.24	205.76	-	83.78
				t_{avg}	1.948	259.050	-	-	139.865
285_300_0.10_0.75	11568	9421	44274.85	f_{best}	11158	11374	*11568*	11391	*11568*
				f_{avg}	10775.9	11143.69	11317.99	10816	**11568**
				std	116.80	76.90	182.82	-	0
				t_{avg}	5.450	426.680	-	24.539	25.128
285_300_0.15_0.85	11517	7634	51440.30	f_{best}	10528	10822	11517	-	**11802**
				f_{avg}	9897.92	10396.60	10899.20	-	**11790.43**
				std	186.53	128.63	300.36	-	27.51
				t_{avg}	5.571	192.575	-	-	206.422
385_400_0.10_0.75	10483	9591	59917.77	f_{best}	10085	10110	10483	9739	**10600**
				f_{avg}	9537.5	9926.18	10013.43	9240	**10536.53**
				std	184.62	87.43	202.40	-	56.08
				t_{avg}	13.012	203.870	-	57.000	234.475
385_400_0.15_0.85	10338	5810	73409.01	f_{best}	9456	9659	10338	-	**10506**
				f_{avg}	9090.03	9444.34	9524.98	-	**10502.64**
				std	156.69	46.40	286.16	-	23.52
				t_{avg}	13.724	177.910	-	-	129.505
485_500_0.10_0.75	11094	5940	84239.56	f_{best}	10823	10835	11094	10539	**11321**
				f_{avg}	10483.4	10789.57	10687.62	10190	**11306.47**
				std	228.34	27.29	168.06	-	36.00
				t_{avg}	27.227	299.260	-	114.066	207.118
485_500_0.15_0.85	10104	4325	100374.77	f_{best}	9333	9380	10104	-	**10220**
				f_{avg}	9085.57	9258.82	9383.28	-	**10179.45**
				std	115.62	58.72	241.01	-	46.97
				t_{avg}	28.493	49.170	-	-	238.630

CPLEX (LB=UB), which are indeed not challenging for the other algorithms. Compared to the reference algorithms (BABC/BABC*, gPSO, MS), I2PLS reports better or equal f_{best} values for all the tested instances without exception. In terms of the average results (f_{avg}), I2PLS also

Table 6
Summary of numbers of instances for which each algorithm reports a better, equal or worse f_{bst} value compared to the best-known value in the literature and *p-values* of the Wilcoxon singned-rank test on f_{best} values over all instances between I2PLS and each reference algorithm including the best-known values.

Instance	Best_Known	BABC	BABC*	gPSO	MSO4	I2PLS
# Better	-	0	2	0	0	18
# Equal	-	2	6	28	3	12
# Worse	-	28	22	2	12	0
p-value	2.14e-4	4.00e-6	2.89e-5	1.43e-4	2.52e-3	-

performs very well by reporting better or equal f_{best} values for all instances except three cases (100_85_0.15_0.85, 100_100_0.15_0.85 and 85_100_0.15_0.85) for which BABC* has better values. Moreover, I2PLS has smaller standard deviations of its f_{best} values (f_{best} values often better than the compared results), suggesting that our algorithm is highly robust.

The small *p-values* (< 0.05) of Table 6 from the Wilcoxon signed-rank test (2.14e-4, 4.00e-6, 2.89e-5 and 1.43e-4) confirm that the results of our algorithm are significantly better than those of the compared results (best known in the literature, BABC, BABC* and gPSO).

Finally, we complete the above presentation by showing graphical comparisons of I2PLS against BABC, BABC*, and gPSO on the three sets of 30 instances. We ignore MS of [9] since no result is available for half of the 30 instances. The plots of Fig. 1 concern the best and average objective values of the compared algorithms while the plots of Fig. 2 are based on the standard deviations. These figures clearly indicate the dominance of the proposed I2PLS algorithm over the reference algorithms in terms of the considered indicators.

5 Analysis and Insights

In this section, we perform an analysis of the parameters and the ingredients of the algorithm to get useful insights about their impacts on its performance.

5.1 Analysis of Parameters

As shown in Table 2, I2PLS requires four parameters: exploration depth λ_{max} (Section 2), neighborhood sampling probability ρ (Section 3.4.1), tabu search depth ω_{max} (Section 3.4.3), perturbation strength η (Section 3.5). To analyze the sensibility and tuning of the parameters, we select 8 out of the 30 benchmark instances, i.e., 185_200_0.15_0.85, 200_185_0.15_0.85,

Fig. 1. The best objective values (left) and mean objective values (right) of BABC, BABC*, gPSO and I2PLS for solving three sets of instances.

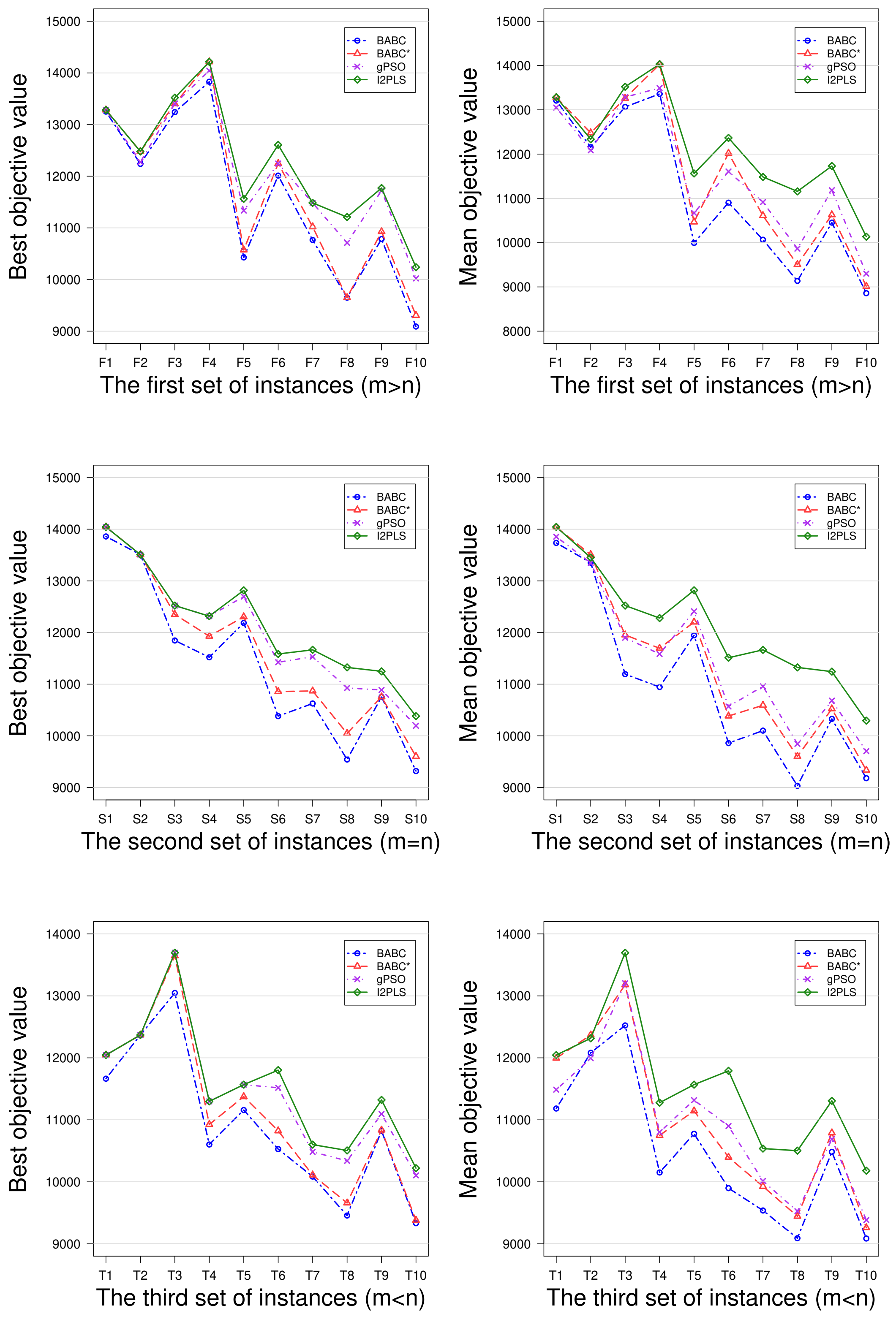
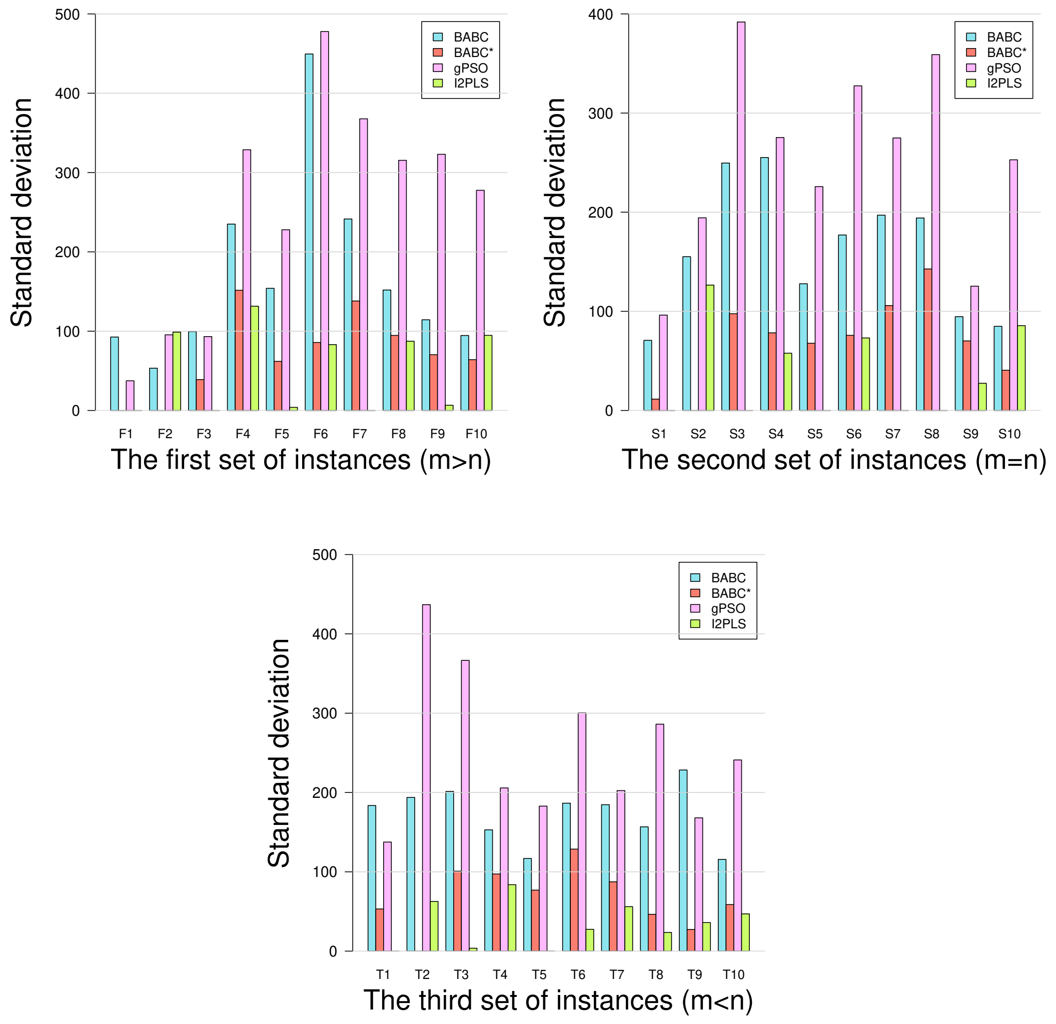


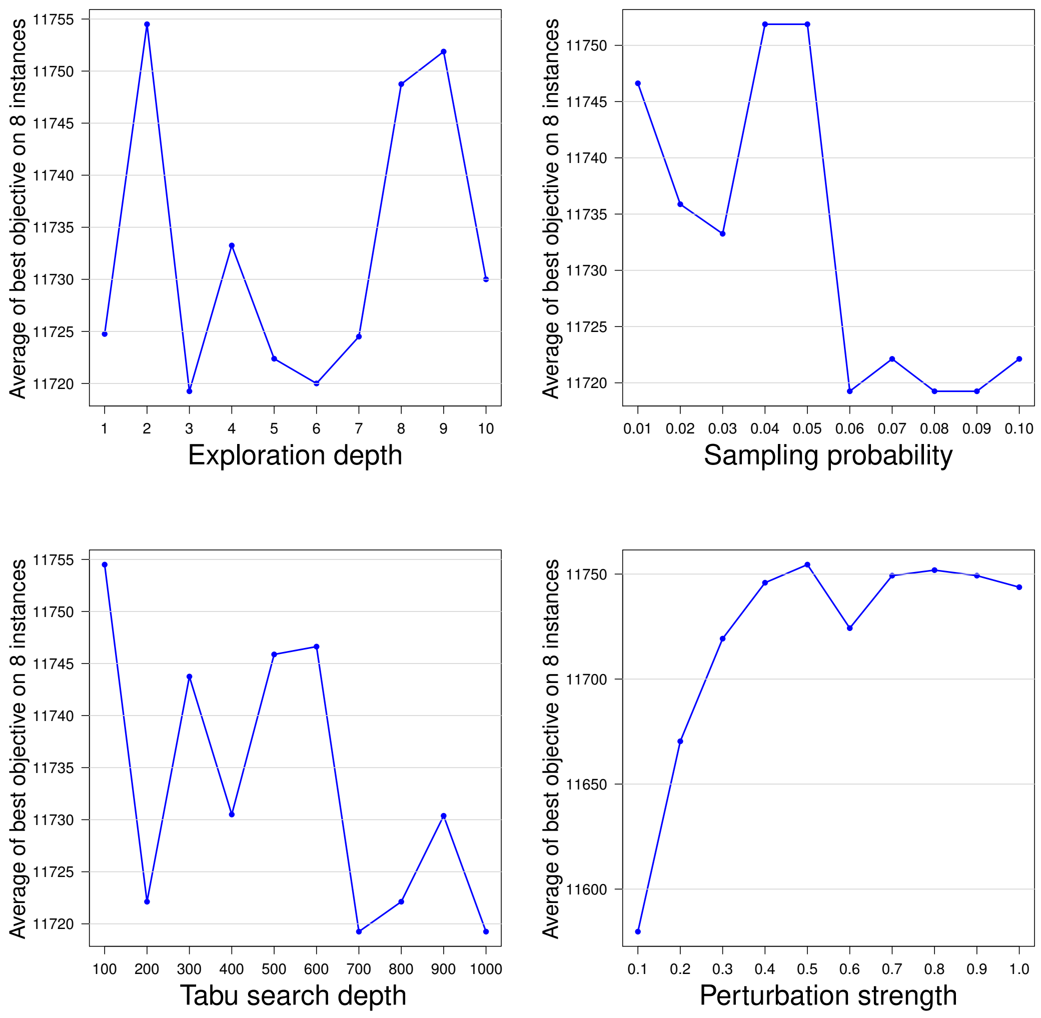
Fig. 2. The standard deviations of BABC, BABC*, gPSO and I2PLS for solving three sets of instances.



200_200_0.15_0.85, 300_285_0.15_0.85, 400_385_0.15_0.85, 500_485_0.10_0.75, 500_485_0.15_0.85 and 500_500_0.15_0.85. According to Tables 3-5, the compared algorithms have a larger standard deviation for most of these instances than for other instances, implying that they are rather difficult to solve. We exclude the instances with 85 and 100 items since they can be solved exactly by the CPLEX and are thus too easy to be used for our analysis.

In this experiment, we studied each parameter independently by varying its value in a pre-determined range while fixing the other parameters to the default values shown in Table 2. We then ran I2PLS with each parameter setting 30 times to solve each of the 8 selected instances with the same cut-off time as in Section 4.3. Specifically, the exploration depth λ_{max} takes its values in $\{1, 2, \ldots, 10\}$ with a step size of 1, the sampling probability ρ

Fig. 3. Average of the best objective values (f_{best}) on the 8 instances obtained by executing I2PLS with different values of the four parameters.



varies from 0.01 to 0.10 with a step size of 0.01, the tabu search depth ω_{max} takes its values in $\{100, 200, \ldots, 1000\}$ with a step size of 100, and the perturbation strength η varies from 0.1 to 1 with a step size of 0.1. Fig. 3 shows the average of the best objective values (f_{best}) obtained by I2PLS with the four parameters on the 8 instances.

Fig. 3 indicates I2PLS achieves better results when $\lambda_{max} = 2, \rho = 0.05$ (the f_{avg} value is better when $\rho = 0.05$ than $\rho = 0.04$), $\omega_{max} = 100, \eta = 0.5$, respectively. This justifies the adopted settings of parameters as shown in Table 2. In addition, for each parameter, we used the non-parametric Friedman test to compare the f_{best} values reached with each of the alternative parameter values. The resulting $p-value$ (> 0.05) of the parameters λ_{max} and ω_{max} show that the differences from alternative parameter settings are not statistically significant, implying that I2PLS is not sensitive to these two parameters.

5.2 Effectiveness of the VND Search Strategy

The VND procedure explores two neighborhoods N_1 and N_2 with a sampling probability ρ applied to N_2. To investigate the impact of this sampling strategy, we performed an experiment by setting $\rho \in \{0.05, 0.0, 1.0\}$, where $\rho = 0.05$ is the adopted value as shown in Table 2, $\rho = 0.0$ indicates that only the neighborhood N_1 is used during the descent search while N_2 is disabled, and $\rho = 1.0$ indicates that the entire neighborhoods N_1 and N_2 are explored.

We denote these three VND variants by $\text{VND}_{0.05}$, $\text{VND}_{0.0}$ and $\text{VND}_{1.0}$ respectively. Recall that the VND procedure adopts the *best-improvement* strategy at each iteration. However, it is interesting to observe the effect of adopting the $first$-*improvement* strategy in N_2, So we included a fourth VND variant with the $first$-*improvement* strategy and $\rho = 1.0$ (denoted as $\text{VND}^{f}_{1.0}$). We ran these four VND variants to solve the 30 benchmark instances under the condition of Section 4.3 and report the results in terms of f_{best} in Table 7 (the best of the f_{best} values in bold). The rows $\#Better$, $\#Equal$ and $\#Worse$ respectively indicate the number of instances for which $\text{VND}_{0.0}$, $\text{VND}_{1.0}$ and $\text{VND}^{f}_{1.0}$ attain a better, equal and worse result compared to the result obtained by $\text{VND}_{0.05}$ (which is the default strategy of I2PLS).

Table 7 shows that $\text{VND}_{0.05}$ performs the best with the setting $\rho = 0.05$. Compared to $\text{VND}_{0.05}$, $\text{VND}_{0.0}$ obtains worse results on 3 instances, and equal results on the other 27 instances. $\text{VND}_{1.0}$ reaches the same results as $\text{VND}_{0.05}$ on 25 instances, and worse results on 5 instances. $\text{VND}_{1.0^f}$ obtains worse results on 4 instances, and equal results on the other 26 instances. Moreover, we observe that when exploring the whole neighborhood N_2, neither the *best-improvement* strategy nor the $first$-*improvement* strategy performs well. This can be explained by the fact that given the large size of N_2, a thorough examination of this neighborhood becomes very expensive. Within the cut-off time, the VND search cannot perform many iterations, decreasing its chance of encountering high-quality solutions. Finally, the $p - value$ of 4.18e-2 from the Friedman test indicates a significant difference among the compared VND strategies. This implies that the adopted VND strategy and sampling technique of the I2PLS algorithm are relevant for its performance.

Table 7
Influence of the VND search strategy on the performance of the I2PLS algorithm.

Instance/Setting	VND$_{0.05}$	VND$_{0.0}$	VND$_{1.0}$	VND$^{f}_{1.0}$
100_85_0.10_0.75	13283	13283	13283	13283
100_85_0.15_0.85	12479	12479	12479	12479
200_185_0.10_0.75	13521	13521	13521	13521
200_185_0.15_0.85	14215	14215	14215	14215
300_285_0.10_0.75	11563	11563	11563	11563
300_285_0.15_0.85	**12607**	12500	12332	12332
400_385_0.10_0.75	11484	11484	11484	11484
400_385_0.15_0.85	11209	11209	11209	11209
500_485_0.10_0.75	**11771**	11729	11746	11729
500_485_0.15_0.85	**10238**	10194	10194	10194
100_100_0.10_0.75	14044	14044	14044	14044
100_100_0.15_0.75	13508	13508	12238	13508
200_200_0.10_0.75	12522	12522	12522	12522
200_200_0.15_0.85	12317	12317	12317	12317
300_300_0.10_0.75	12817	12817	12817	12817
300_300_0.15_0.85	11585	11585	11502	11585
400_400_0.10_0.75	11665	11665	11665	11665
400_400_0.15_0.85	11325	11325	11325	11325
500_500_0.10_0.75	11249	11249	11249	11249
500_500_0.15_0.85	10381	10381	10381	10381
85_100_0.10_0.75	12045	12045	12045	12045
85_100_0.15_0.85	12369	12369	12369	12369
185_200_0.10_0.75	13696	13696	13696	13696
185_200_0.15_0.85	11298	11298	11298	11298
285_300_0.10_0.75	11568	11568	11568	11568
285_300_0.15_0.85	11802	11802	11802	11802
385_400_0.10_0.75	10600	10600	10600	10600
385_400_0.15_0.85	10506	10506	10506	10506
485_500_0.10_0.75	11321	11321	11321	11321
485_500_0.15_0.85	10220	10220	10220	10208
# Better	-	0	0	0
# Equal	-	27	25	26
# Worse	-	3	5	4

5.3 Effectiveness of the Frequency-Based Local Optima Escaping Strategy

The frequency-based local optima escaping strategy of I2PLS perturbs the locally best solution $S_b = (A, \bar{A})$ by replacing the first $\eta \times |A|$ (in I2PLS, η is set to 0.5) least frequently moved items of A with items that are randomly chosen from $\bar{A}$. In this experiment, we compared I2PLS against two variants with alternative perturbation strategies. In the first variant (denoted by I2PLS$_{random}$), we replace $0.5 \times |A|$ items randomly selected items of A while in the second variant (denoted by I2PLS$_{strong}$) and we perform a very strong perturbation by replacing all the items of A with items of $\bar{A}$ (i.e., setting η to 1). We ran I2PLS, I2PLS$_{random}$ and I2PLS$_{strong}$ 30 times to solve each of the 30 benchmark instances. The computational results of this experiment are shown in Table 8 where in addition to the best f_{best} values of each compared algorithm (the best of the f_{best} values in bold), the last three rows indicate the number of instances for which I2PLS$_{random}$ and I2PLS$_{strong}$ has a better, equal and worse result compared to that of I2PLS.

Table 8
Impact of the frequency-based local optima escaping strategy on the performance of the I2PLS algorithm.

Instance/Setting	I2PLS	I2PLS$_{random}$	I2PLS$_{strong}$
100_85_0.10_0.75	13283	13283	13283
100_85_0.15_0.85	12479	12479	12479
200_185_0.10_0.75	13521	13521	13521
200_185_0.15_0.85	14215	14215	14215
300_285_0.10_0.75	11563	11563	11563
300_285_0.15_0.85	12607	12607	12607
400_385_0.10_0.75	11484	11484	11484
400_385_0.15_0.85	11209	11209	11209
500_485_0.10_0.75	**11771**	11729	11729
500_485_0.15_0.85	**10238**	10194	10194
100_100_0.10_0.75	14044	14044	14044
100_100_0.15_0.75	13508	13508	13508
200_200_0.10_0.75	12522	12522	12522
200_200_0.15_0.85	12317	12317	12317
300_300_0.10_0.75	12817	12817	12817
300_300_0.15_0.85	11585	11585	11585
400_400_0.10_0.75	11665	11665	11665
400_400_0.15_0.85	11325	11325	11325
500_500_0.10_0.75	11249	11249	11249
500_500_0.15_0.85	10381	10381	10381
85_100_0.10_0.75	12045	12045	12045
85_100_0.15_0.85	12369	12369	12369
185_200_0.10_0.75	13696	13696	13696
185_200_0.15_0.85	11298	11298	11298
285_300_0.10_0.75	11568	11568	11568
285_300_0.15_0.85	11802	11802	11802
385_400_0.10_0.75	10600	10600	10600
385_400_0.15_0.85	10506	10506	10506
485_500_0.10_0.75	11321	11321	11321
485_500_0.15_0.85	10220	10220	10220
# Better	-	0	0
# Equal	-	28	28
# Worse	-	2	2

Table 8 shows that I2PLS with its frequency-based local optima escaping strategy performs slightly better than the two variants with alternative perturbation strategies. Indeed, even if the compared strategies lead to equal results for 28 instances, I2PLS achieves a better result on two of the most difficult instances (500_485_0.10_0.75 and 500_485_0.15_0.85). This experiment tends to indicate that the frequency-based local optima escaping strategy is particularly helpful for solving difficult instances. The $p - value$ of 1.35e-1 from the Friedman test indicates that the compared strategies differ only marginally.

6 Conclusions

The set-union knapsack problem (SUKP) studied in this work is a generalization of the conventional 0-1 knapsack problem with a variety of practical applications. Existing solution methods are mainly based on swarm optimization. This work introduces the first local search approach for solving the SUKP that directly operates in the discrete search space. The proposed algorithm combines a local optima exploration phase and a local optima escaping phase based on frequency information within the iterated local search framework.

The proposed algorithm has been tested on three sets of 30 benchmark instances commonly tested in the literature and showed a high competitive performance compared to the state-of-the-art SUKP algorithms. Specifically, our algorithm has improved on the best-known results (new lower bounds) for 18 out of the 30 benchmark instances, while matching the best-known results for the remaining 12 instances. Moreover, we has investigated for the first time the interest of the general mixed integer linear programming solver CPLEX for solving the SUKP, showing that the optimal solutions can be reached only for 6 small instances. Furthermore, we have analyzed the impacts of parameters and the main components of the algorithm on its performance.

This work can be further improved. First, even if the algorithm uses the filtering mechanism and the sampling technique to reduce the neighborhoods, evaluating a given neighbor solution remains time-comsuming. To speed up the search process, it is useful to seek streamlining techniques to reduce the complexity of neighborhood evaluation. Second, considering the potential strong correlations of constituent elements between different items, a hybrid approach combining local search and population-based search could be helpful to break search barriers and traps. Finally, the SUKP belongs to the large family of knapsack problems, it would be interesting to investigate whether proven techniques and strategies designed for related knapsack problems remain useful for solving the SUKP.

Acknowledgments

We are grateful to the reviewers for their insightful comments that helped us to improve the work. We would like to thank Prof. Yichao He and his co-authors for making the benchmark instances tested in [14] available and sharing the codes of their BABC algorithm with us. Support from the ChinaScholarship Council (CSC, grant No: 201706290016) for the first author is also acknowledged.

References

[1] Ashwin Arulselvan. A note on the set union knapsack problem. *Discrete Applied Mathematics*, 169:214–218, 2014.

[2] Mustafa Avci and Seyda Topaloglu. A multi-start iterated local search algorithm for the generalized quadratic multiple knapsack problem. *Computers & Operations Research*, 83:54–65, 2017.

[3] Una Benlic and Jin-Kao Hao. Breakout local search for the quadratic assignment problem. *Applied Mathematics and Computation*, 219(9):4800–4815, 2013.

[4] Eden Chlamtác, Michael Dinitz, Christian Konrad, Guy Kortsarz, and George Rabanca. The densest k-subhypergraph problem. *SIAM Journal on Discrete Mathematics*, 32(2):1458–1477, 2018.

[5] Paola Cappanera and Marco Trubian. A local-search-based heuristic for the demand-constrained multidimensional knapsack problem. *INFORMS Journal on Computing*, 17(1):82–98, 2005.

[6] Yuning Chen and Jin-Kao Hao. A "reduce and solve" approach for the multiple-choice multidimensional knapsack problem. *European Journal of Operational Research*, 239(2):313–322, 2014.

[7] Yuning Chen and Jin-Kao Hao. Iterated responsive threshold search for the quadratic multiple knapsack problem. *Annals of Operations Research*, 226(1):101–131, 2015.

[8] Yuning Chen and Jin-Kao Hao. An iterated "hyperplane exploration" approach for the quadratic knapsack problem. *Computers & Operations Research*, 77:226–239, 2017.

[9] Yanhong Feng, Haizhong An, and Xiangyun Gao. The importance of transfer function in solving set-union knapsack problem based on discrete moth search algorithm. *Mathematics*, 7(1):17, 2019.

[10] Zhanghua Fu and Jin-Kao Hao. A three-phase search approach for the quadratic minimum spanning tree problem. *Engineering Applications of Artificial Intelligence*, 46:113–130, 2015.

[11] Fred Glover and Gary A Kochenberger. Critical event tabu search for multidimensional knapsack problems. In *Meta-Heuristics*, pages 407–427. Springer, 1996.

[12] Fred Glover and Manuel Laguna. Tabu Search. Springer US, 1997.

[13] Olivier Goldschmidt, David Nehme, and Gang Yu. Note: On the set-union knapsack problem. *Naval Research Logistics*, 41(6):833–842, 1994.

[14] Yichao He, Haoran Xie, Tak-Lam Wong, and Xizhao Wang. A novel binary artificial bee colony algorithm for the set-union knapsack problem. *Future Generation Computer Systems*, 78:77–86, 2018.

[15] Holger H. Hoos and Thomas Stützle. Stochastic Local Search - Foundations and Applications. Elsevier, 2004.

[16] Mhand Hifi, Mustapha Michrafy, and Abdelkader Sbihi. A reactive local search-based algorithm for the multiple-choice multi-dimensional knapsack problem. *Computational Optimization and Applications*, 33(2-3):271–285, 2006.

[17] Hans Kellerer, Ulrich Pferschy, and David Pisinger. Knapsack problems. Springer, 2004.

[18] Xiangjing Lai, Jin-Kao Hao, Fred Glover, and Zhipeng Lü. A two-phase tabu-evolutionary algorithm for the 0-1 multidimensional knapsack problem. *Information Sciences*, 436:282–301, 2018.

[19] Xiangjing Lai, Jin-Kao Hao, and Dong Yue. Two-stage solution-based tabu search for the multidemand multidimensional knapsack problem. *European Journal of Operational Research*, 274(1):35–48, 2018.

[20] Helena R Lourenço, Olivier C Martin, and Thomas Stützle. Iterated local search. In *Helena R. Lourenço, Olivier C. Martin, and Thomas Stützle (Eds). Handbook of Metaheuristics, International Series in Operations Research & Management Science book series*, volume 57, pages 320–353, 2003.

[21] Nenad Mladenoviç and Pierre Hansen. Variable neighborhood search. *Computers and Operations Research*, 24(11):1097–1100, 1997.

[22] Fehmi B Ozsoydan and Adil Baykasoglu. A swarm intelligence-based algorithm for the set-union knapsack problem. *Future Generation Computer Systems*, 93:560–569, 2019.

[23] Bo Peng, Mengqi Liu, Zhipeng Lü, Gary Kochenberger, Haibo Wang. An ejection chain approach for the quadratic multiple knapsack problem. *European Journal of Operational Research*, 253(2):328–336, 2016.

[24] Richard Taylor. Approximations of the densest k-subhypergraph and set union knapsack problems. *arXiv preprint arXiv:1610.04935*, 2016.

[25] Michel Vasquez and Jin-Kao Hao. A hybrid approach for the 0-1 multidimensional knapsack problem. In *Proceedings of 17th International Joint Conference on Artificial Intelligence-IJCAI'2001*, 4-10 August, 2001, Seattle, Washington, USA, pages 328–333.

[26] Qinghua Wu and Jin-Kao Hao. A review of algorithms for maximum clique problems. *European Journal of Operational Research*, 242(3):693–709, 2015.

[27] Zhen Yang, Guoqing Wang, and Feng Chu. An effective grasp and tabu search for the 0–1 quadratic knapsack problem. *Computers and Operations Research*, 40(5):1176–1185, 2013.

[28] Yi Zhou, Jin-Kao Hao, and Adrien Goëffon. PUSH: a generalized operator for the maximum weight clique problem. *European Journal of Operational Research*, 257(1):41–54, 2017.

A Appendix: 0/1 linear programming model for the SUKP

We present the mathematical model that we proposed and used to report the results (lower and upper bounds) of the general ILP solver CPLEX in Section 4.3. Our model is based on the mathematical model of the SUKP introduced in [14] (see the detailed description of this model in Section 2, page 78 of this reference), which is, however, inapplicable by the CPLEX solver. We introduce below the modified 0/1 linear programming model that is suitable for the solver. For an arbitrary non-empty item set $S \subset V$ represented by its binary vector $S = (y_1, \ldots, y_m)$ such that $y_i = 1$ $(i = 1, \ldots, m)$ if item i is selected in S, and $y_i = 0$ otherwise. Let R be a $m \times n$ binary relation matrix such that $R_{ij} = 1$ if element j belongs to item i, and $R_{ij} = 0$ otherwise. Furthermore, for each element j $(j = 1, \ldots, n)$, define $L_j = \sum_{i=1}^{m} y_i R_{ij}$ that counts the number of appearances of element j in the items of S. Let x_j be a binary variable such that $x_j = 1$ if $L_j > 0$, and $x_j = 0$ otherwise, that is, x_j indicates whether element j is involved in calculating the total weight of S. Then our 0/1 linear programming model for the SUKP is defined as follows.

$$(SUKP) \quad \text{Maximize} \quad f(S) = \sum_{i=1}^{m} p_i y_i \tag{A.1}$$

$$\text{s.t.} \quad W(S) = \sum_{j=1}^{n} w_j x_j \leq C \tag{A.2}$$

$$x_j = \begin{cases} 1, & if\ L_j > 0; \\ 0, & \text{otherwise.} \end{cases} \tag{A.3}$$

$$L_j = \sum_{i=1}^{m} y_i R_{ij}, j = 1, \ldots, n \tag{A.4}$$

$$y_i \in \{0, 1\}, i = 1, \ldots, m \tag{A.5}$$

Constraints A.2–A.4 jointly ensure that the weight w_j of an element j is counted only once in $W(S)$ even if the element appears in more than one selected items and the capacity constraint is satisfied. Constraints A.5 guarantee that each item is selected at most once. Equation (A.1) maximizes the total profit of the selected items.