Version of Record: https://www.sciencedirect.com/science/article/pii/S0957417420306199
Manuscript_31b21dc36d2ff5951ecffd45f78be5eb

Kernel based Tabu Search for the Set-Union Knapsack Problem

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

[a] *LERIA, Université d'Angers, 2 bd Lavoisier, 49045 Angers, Cedex 01, France*
[b] *Institut Universitaire de France, 1 Rue Descartes, 75231 Paris, France*

Abstract

Given a set of profitable items where each item is a set of weighted elements, the Set-union Knapsack Problem is to pack a subset of items into a capacity constrained knapsack to maximize the total profit of the selected items. This problem appears in many practical applications; however, it is computationally challenging. To advance the state-of-the-art for solving this relevant problem, we introduce a competitive heuristic algorithm, which features original kernel-based search components and an effective local search procedure. Extensive computational assessments on 60 benchmark instances demonstrate the high performance of the algorithm. We show different analyses to get insights into the influences of its algorithmic components. We make the code of the algorithm publicly available to facilitate its use in practice.

Keywords: Knapsack; Heuristics and metaheuristics; Decision making; Intelligent systems; Combinatorial optimization.

1. Introduction

As a generalized knapsack model, the Set-Union Knapsack Problem (SUKP) is defined as follows (Kellerer et al., 2004). Given 1) a set U of n elements where each element j has a weight $w_j > 0$, 2) a set V of m items where each item i is a subset of elements $U_i \subseteq U$ and has a profit $p_i > 0$, and 3) a knapsack of capacity C, SUKP involves determining a set of items $S \subseteq V$ to maximize the total profit of S while ensuring that the total weight of the elements of S does not exceed the knapsack capacity C. Notice that the weight of an element is counted only once even if it belongs to more than one selected items in S. Formally, SUKP can be written as follows.

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

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

© 2020 published by Elsevier. This manuscript is made available under the Elsevier user license
https://www.elsevier.com/open-access/userlicense/1.0/

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

Like other knapsack models (Amiri, 2020; Dahmani et al., 2020; Denysiuk et al., 2019; Glover & Kochenberger, 1996; Qin et al., 2016; Lai et al., 2018b; Vasquez & Hao, 2001), SUKP has a number of practical applications. As an example, we consider the following decision-making problem to optimally allocate data in large cyber systems (Tu & Xiao, 2016). Given a centralized cyber system with a memory of fixed capacity holding a set of services (or requests) with profits, where each service contains a set of data objects. Each data object will consume a certain amount of memory when it is invoked, and multiple use of the same data object will not cause additional memory consumption. The goal is to select a subset of services, among the candidate services, such that the total profit of the selected services is maximized while the total memory consumed by the underlying data objects meets the memory capacity of the cyber system. This application can be conveniently formulated by the SUKP model where an item corresponds to a service with its profit and an element corresponds to a data object with its memory consumption (element weight). Then, solving the data allocation problem is equivalent to find the optimal solution to the resulting SUKP problem. SUKP has other relevant applications related to decision-making and intelligent systems including database partitioning (Navathe et al., 1984), flexible manufacturing (Goldschmidt et al., 1994), key-pose caching (Lister et al., 2010), and public key prototyping (Schneier, 1996).

Meanwhile, in terms of computational complexity theory, the decision version of SUKP is known to be NP-complete (Goldschmidt et al., 1994). Therefore from the perspective of solution methods, solving the problem is a highly challenging task. Given its practical and theoretical relevance, a number of algorithms for SUKP have been introduced in the literature.

First, exact and approximation algorithms based on dynamic programming or greedy approximation methods were investigated in (Goldschmidt et al., 1994; Taylor, 2016; Arulselvan, 2014). These studies are of theoretical nature and didn't show computational results.

Second, given the NP-hardness of SUKP, several algorithms based on metaheuristics were proposed recently to find approximate solutions in a reasonable time frame. He et al. (2018) designed a binary artificial bee colony algorithm (BABC) for solving SUKP and reported the first computational study on a set of 30 benchmark instances they introduced. Later, He and Wang (2018) devised a group theory-based optimization algorithm (GTOA) for several knapsack problems including SUKP. Then, Ozsoydan and Baykasoğlu (2018) presented a binary swarm intelligence algorithm (gPSO) that combines the genetic algorithm with particle swarm optimization. Baykasoğlu et al. (2018) proposed a modified weighted superposition attraction algorithm (WSA) for stationary binary optimization problems including SUKP. Ozsoydan (2019) introduced a swarm-based optimization algorithm (intAgents) using artificial search agents with individual cognitive intelligence. Feng et al. (2019a; 2019b) introduced two moth search algorithms (MS and EMS). These algorithms share the common

feature that they solve the discrete SUKP indirectly by performing their search in a continuous search space. Wei and Hao (2019) presented the first binary optimization method for SUKP with two complementary local search phases (I2PLS). Wu and He (2020) presented a hybrid Jaya algorithm (DHJaya) based on the differential evolution crossover operator and Cauchy mutation strategy. Lin et al. (2019) proposed a hybrid binary particle swarm optimization method (HBPSO/TS). Finally, Liu and He (2019) combined the estimation of distribution algorithm based on Lévy flight (LFEDA) with a quadratic greedy repair and optimization approach.

The literature review shows that the existing algorithms have a number of limitations. First, the performances of these algorithms lack stability and robustness (computational results with large standard deviations) even when solving small benchmark instances (with 85 to 100 items and elements). Second, their performances generally decrease when they are used to solve large instances (with at least 500 items and elements). Third, they consume a substantial amount of computation time to reach their reported results. Finally, most existing algorithms require a non-negligible number of parameters (e.g., 4 and 7 parameters for two leading algorithms I2PLS and HBPSO/TS, respectively), making it difficult to control their performances and understand their behaviors.

In this work, we aim at advancing the state-of-the-art of solving SUKP effectively and robustly in particular when large problem instances are considered. For this purpose, we investigate the first *kernel* based approach that overcomes the limitations mentioned above. This work is also motivated by another important consideration. In fact, the general idea of kernel has proved to be quite useful for several binary optimization problems (e.g., Vasquez & Hao (2001); Wang et al. (2013); Zhang (2004)). This work demonstrates for the first time its benefit for solving SUKP, whose contributions are summarized as follows.

First, to evaluate the meaningfulness of the idea of kernel for solving SUKP, we investigate the distribution of items among high-quality solutions. This investigation reveals the existence of kernels, which lays the basis for adopting the kernel concept to design our search algorithm. Indeed, the proposed kernel based tabu search algorithm (KBTS) integrates three complementary search components to perform an effective examination of the search space. That is, a local search procedure is used to find various local optima, a kernel search method is employed to discover additional high-quality solutions within particular areas, and a non-kernel search method is applied to ensure a guided diversification.

Second, we show the competitiveness of the proposed algorithm by comparing it with the state-of-the-art algorithms on 60 benchmark instances. We provide new lower bounds for several benchmark instances that can contribute to future research on SUKP.

Third, we make the code of our KBTS algorithm publicly available, which can help researchers and practitioners to better solve various problems that can be formulated as SUKP.

Finally, the kernel based search components of the proposed algorithm rely on general principals that can be advantageously adapted to other binary opti-

mization problems.

The rest of the paper is structured as follows. Section 2 presents the proposed algorithm as well as its components. Section 3 shows computational results and comparisons with the state-of-the-art algorithms. Section 4 shows several analyses to shed lights on the understanding of the key ingredients of the algorithm. Conclusions and research perspectives are provided in the last section.

2. Kernel Based Tabu Search for SUKP

In this section, we present the KBTS algorithm for solving SUKP. We first present its main scheme and then describe its components.

2.1. Main scheme

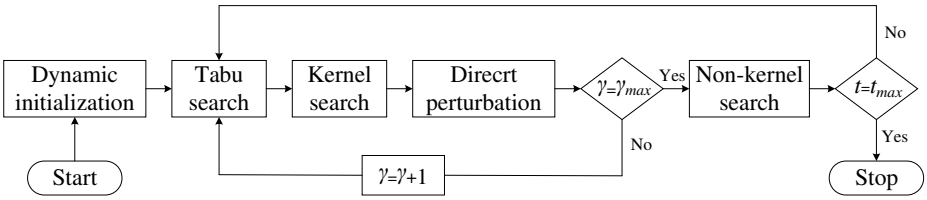

Fig. 1. Flow chart of the KBTS algorithm.

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

The algorithm starts from a feasible initial solution generated by a *dynamic profit-ratio* mechanism (line 3, Alg. 1, and Section 2.3). Then it enters a 'while' loop to execute the main search process. Specifically, the input solution is improved by an iterative process (the 'repeat' loop), which includes a tabu search procedure, a kernel search procedure and a direct perturbation procedure. At each iteration of this process, the tabu search procedure (line 10, Alg. 1) is first invoked to obtain a high-quality solution with the neighborhood N_f (Section 2.4.1). During tabu search, a kernel solution (S_k) as well as a non-kernel solution ($\bar{S}_k$) are created using information from a frequency counter Φ. Then the kernel search procedure (line 11, Alg. 1, and Section 2.5) uses the neighborhood N_k to perform an intensified search around the kernel solution to seek other high-quality solutions. After that, the direct perturbation procedure (Section 2.6) is applied to modify the last local optimum found (controlled by the parameter δ), which is then used to start the next iteration of the process. This process ends when γ_{max} consecutive iterations are reached without further improving the local best solution S_b. At this point, the search is judged to be exhausted with the current search region and switches to the non-kernel search procedure (Section 2.7) to explore a distant and unexplored region. Finally, the whole algorithm terminates when the given time limit t_{max} is reached and returns the overall best solution S^* found during the search.

Algorithm 1 Kernel Based Tabu Search for SUKP

Input: Instance I, cut-off time t_{max}, neighborhoods N_f, N_k, $\bar{N_k}$, *local search depth* γ_{max}, *kernel coefficient* ε, *direct perturbation strength* δ.
Output: The best solution found S^*.
$S \leftarrow$ Dynamic_Initialization(I) /* Generate an initial solution S, Sect. 2.3 */
$S^* \leftarrow S$ /* Record the overall best solution S^**/
while $Time \leq t_{max}$ **do**
  $\Phi \leftarrow$ Frequency_Initialization() /* Initialize frequency counter Φ to 0 */
  $S_b \leftarrow S$ /* Record the best solution S_b found so far */
  $\gamma \leftarrow 0$ /*γ counts the number of consecutive non-improving rounds*/
  repeat
    /* Record the local optimum S_l found by tabu search */
    $(S_l, S_k, \bar{S_k}) \leftarrow$ Tabu_Search $(S, N_f, \Phi, \varepsilon)$ /* Sect. 2.4 */
    $S_l \leftarrow$ Kernel_Search(S_k, S_l, N_k) /* Sect. 2.5 */
    $S \leftarrow$Direct_Perturbation(S_l, δ) /* Sect. 2.6 */
    if $f(S_l) > f(S_b)$ **then**
      $S_b \leftarrow S_l$ /* Update the local best solution S_b found so far */
      $\gamma \leftarrow 0$
    else
      $\gamma \leftarrow \gamma + 1$
    end if
  until $\gamma = \gamma_{max}$
  if $f(S_b) > f(S^*)$ **then**
    $S^* \leftarrow S_b$ /* Update the overall best solution S^* found so far */
  end if
  $S \leftarrow$ Non-Kernel_Search($\bar{S_k}, \bar{N_k}$) /* Sect. 2.7 */
end while
return S^*

2.2. Solution representation, search space, and evaluation function

The search of the KBTS algorithm is limited to the feasible solution space Ω^F satisfying the knapsack constraint. By reference to the item set V with m items, a candidate solution S of Ω^F can be conveniently represented by $S = (y_1, \ldots, y_m)$ where each y_i is a binary variable: $y_i = 1$ if item i is selected, $y_i = 0$ otherwise. A solution S can also be represented by $S =< A, \bar{A} >$ where $A \subseteq V$ is the set of selected items and $\bar{A} = V \setminus A$ is the set of the remaining items. The quality of S is measured by its objective value $f(S) = \sum_{i=1}^{m} p_i y_i$.

2.3. Dynamic initialization

The KBTS algorithm adopts an original initialization procedure using a *dynamic profit-ratio* of non-selected items. This procedure is based on the fact that for a given solution S, the weight of each element is counted only once. When a new item k is added to S, only the new elements of k that do not belong to the subset S will impact the total weight. Therefore, in our initialization procedure, the *profit-ratio* of non-selected items will be recalculated according to the elements belonging to the current solution S after adding a new item into S. The *dynamic profit-ratio* r_k^* of a non-selected item k is then given by $r_k^* = p_k / \sum_{j \in U_k \wedge j \notin \cup_{i \in S} U_i} w_j$.

From an empty subset S, the dynamic initialization procedure operates as follows. First, we calculate the *dynamic profit-ratio* r_k^* of non-selected items. Second, we identify the item k with the highest r_k^* value and add the item into S. We iterate these two steps until the knapsack constraint is reached.

Note that the *dynamic profit-ratio* refines the *static profit-ratio* used in (Wei & Hao, 2019) and generally leads to solutions of better quality.

2.4. Tabu search procedure

The KBTS algorithm adopts the well-known tabu search (TS) metaheuristic (Glover & Laguna, 1997) to explore local optima within a restricted neighborhood. As a *general* search method, TS needs to be adequately adapted to the specific optimization problem under consideration. One notices that TS is quite successful to solve several knapsack problems (e.g., quadratic multiple knapsack (Qin et al., 2016), multidimensional knapsack (Glover & Kochenberger, 1996; Lai et al., 2018b), set-union knapsack problem (Lin et al., 2019; Wei & Hao, 2019)) and other optimization problems (e.g., Díaz et al. (2017); Lai et al. (2020)).

Our tabu search procedure is shown in algorithm 2, whose particular features tailored to SUKP are discussed below. Given an input solution S, the TS procedure explores the neighborhood $N_f(S)$ induced by the *swap* operator (see Section 2.4.1) to make transitions from the current solution to neighbor solutions. Specifically, for each 'while' iteration (lines 5-11, Alg. 2), TS selects the best neighbor solution with the neighborhood search procedure, which is shown in Algorithm 3. If the new selected solution S is better than the best solution S_l found during tabu search, S_l is updated by S. Meanwhile, the frequency counter Φ_i of each selected item i in S is updated by $\Phi_i = \Phi_i + 1$, The main search ('while' loop) terminates when the neighborhood $N_f(S)$ becomes empty (see Algorithm 3). Then the kernel solution S_k and non-kernel solution $\bar{S_k}$ are created based on the frequency counter Φ, which will be presented in Sections 2.5 and 2.7.

Algorithm 2 Tabu Search

1: **Input**: Input solution S, neighborhood N_f, frequency counter Φ, *kernel coefficient* ε.
2: **Output**: Best solution S_l found during tabu search, kernel solution S_k, non-kernel solution $\bar{S_k}$.
3: $S_l \leftarrow S$ /* Record the best solution S_l found during tabu search */
4: $Continue \leftarrow True$
5: **while** $Continue$ **do**
6: $(Continue, S) \leftarrow$ Neighborhood_Search(S, N_1, $Continue$) /* Algorithm 3 */
7: **if** $f(S) > f(S_l)$ **then**
8: $S_l \leftarrow S$ /* Update the best solution found during tabu search */
9: $\Phi \leftarrow$ Update_Frequency(Φ)
10: **end if**
11: **end while**
12: $S_k \leftarrow$ Create_Kernel(Φ, ε)
13: $\bar{S_k} \leftarrow$ Create_Non_Kernel(S_k)
14: **return** $(S_l, S_k, \bar{S_k})$

Algorithm 3 Neighborhood Search

Input: Input solution S, flag $Continue$, neighborhood N.
Output: $Continue$, best solution S found.
Find admissible neighbor solutions $N(S)$
if $N(S) \neq \varnothing$ **then**
    $S \leftarrow argmax\{f(S') : S' \in N(S)\}$ /* Attain the best neighbor solution S */
    $Update\ tabu\_list$
    $Continue = True$
else
    $Continue = False$
end if
return $(Continue, S)$

2.4.1. Move operator and neighborhood structure

From the current solution, a neighbor solution is generated by applying the popular *swap* operator (Wei & Hao, 2019). Specifically, given a solution $S =< A, \bar{A} >$ where $A \subseteq V$ is the set of selected items and $\bar{A} = V \setminus A$, a $swap(q, p)$ operation exchanges q items in A with p items in $\bar{A}$, leading to a neighbor solution designated by $S \oplus swap(q, p)$. Note that q and p refer to the number of items involved in the *swap* operator. In our case, the candidate values for q and p are 0 or 1. Therefore, the *swap* operator includes three different operations: the *Add* operation with $q = 0$ and $p = 1$ (add one item from $\bar{A}$ into A), the *Delete* operation with $q = 1$ and $p = 0$ (delete one item from A) and the *Exchange* operation with $q = 1$ and $p = 1$ (exchange one item of A against one item of $\bar{A}$). Then the basic neighborhood induced by the *swap* operator includes all feasible solutions obtained by $S \oplus swap(q, p)$.

To enhance the computational efficiency of the KBTS algorithm, we define a restricted neighborhood by using a neighborhood filtering strategy (Wei & Hao, 2019; Lai et al., 2018a) to exclude unpromising neighbor solutions. With this strategy, only neighbor solutions S' of reasonable quality verifying $f(S') > f(S_b)$ are considered where S_b is the best solution found so far in the current tabu search run. Formally, the filter-based neighborhood $N_f(S)$ is defined as follows.

$$N_f(S) = \{S' : S' = S \oplus swap(q, p), q \in \{0, 1\}, p \in \{0, 1\}, f(S') > f(S_b)\} \quad (3)$$

Furthermore, to ensure the computational efficiency when evaluating a feasible neighbor solution, we adopt the so-called *gain updating* strategy (Lin et al., 2019; Wei & Hao, 2019). Specifically, we use a vector G of length n where G_j ($G_j \in \{0, 1 \ldots, n\}$) records the number of appearances of element j in a solution S. Thus, only the elements that change values in G after performing $swap(q, p)$ will be considered when calculating the total weight of a new neighbor solution $S \oplus swap(q, p)$. That is, for each element j, if its G_j value changes from zero to non-zero, the total weight of the new solution is increased by w_j; if G_j changes from non-zero to zero, then total weight of the new solution is decreased by w_j. In other cases, the weight of the neighbor solution remains unchanged.

2.4.2. Tabu list management and aspiration criterion

Our TS procedure employs a tabu list to avoid revisiting previous encountered solutions. When a *swap* operation is performed, each item i involved in the swap is added in the tabu list and forbidden to move away from their respective item set for the next T_i consecutive iterations, where T_i is called the tabu tenure. Inspired by the tabu list management proposed in (Vasquez & Hao, 2001), our tabu tenure T_i is set to the number of times item i is moved by the *swap* operation. As such, items with a high (low) move frequency will be forbidden for a longer (shorter) time. When no admissible move is available in the neighborhood (i.e., $N_f(S) = \emptyset$), the TS procedure automatically stops.

During the tabu search, a best neighbor solution among those that are allowed by the tabu list is selected to replace the current solution. Notice that a neighbor solution is always selected if it is better than the best solution found during the TS procedure even if the solution is forbidden by the tabu list. This is the so-called *aspiration criterion* in tabu search (Glover & Laguna, 1997).

2.5. Kernel search procedure

The tabu search procedure is able to explore different local optimal solutions with the help of the tabu list. Still, some interesting zones with better solutions may be overlooked. The kernel search procedure is introduced to perform an additional examination of particular regions identified by the so-called kernel solution.

Definition 1. *Let $\mathcal{S}$ be a set of feasible solutions, k an integer, and Φ_i the frequency of item i appearing in the solutions of $\mathcal{S}$, then the kernel solution (or simply kernel) S_k is the set of top k items with the highest frequencies such that $\Phi_i \geq \Phi_k$ and the total weight of S_k does not exceed the knapsack capacity.*

In the KBTS algorithm, we employ the frequency counter Φ_i to keep track of the number of times each item i appears in high-quality solutions. As mentioned in Section 2.4 (line 9, Alg. 2), each time a better solution is found during the tabu search procedure, the frequency counter Φ_i of the selected item i is updated by $\Phi_i = \Phi_i + 1$. Then at the end of the TS procedure, we generate the kernel S_k in two steps (line 12, Alg. 2). First, we sort all items in descending order according to the values of Φ. Second, we add the top $\varepsilon \times |S_l|$ most frequently appearing items to S_k, where ε is a parameter called *kernel coefficient* and $|S_l|$ is the number of the selected items in the best solution found during tabu search. Then S_k serves as the input solution S for the kernel search (KS) procedure shown in Algorithm 4.

The kernel search procedure shares the same framework with the TS procedure and employs the same neighborhood search procedure (see Algorithm 3), the same tabu list management and aspiration criterion. However, the KS procedure performs its search with the kernel based neighborhood $N_k(S)$ which is composed of neighbor solutions induced by the swap operator applied to the items of S excluding those of the kernel S_k. In other words, the items belonging to the kernel S_k remain fixed during the kernel search and do not take part in

any swap operation. By freezing the items of the kernel during the search, the KS procedure ensures a strongly intensified examination around the kernel.

The KS procedure ends if no admissible move is available in the kernel based neighborhood $N_k(S)$. At this point, the region around the kernel is considered to be sufficiently examined and the algorithm needs to move to a new region to continue its search. For this, we employ a direct perturbation strategy that is explained in the next section.

Algorithm 4 Kernel Search

```
1: Input: Input kernel solution S_k, attained local optimum S_l, neighborhood N_k.
2: Output: Best solution S_l during kernel search.
3: S ← S_k                                          /* Generate a new solution by S_k */
4: Continue ← True
5: while Continue do
6:    (Continue, S) ← Neighborhood_Search(S, N_k, Continue)
7:    if f(S) > f(S_l) then
8:       S_l ← S                    /* Update the best solution found during kernel search */
9:    end if
10: end while
11: return S_l
```

The kernel search procedure is inspired by the work presented in (Vasquez & Hao, 2001) where the notion of kernel was introduced for solving a logic-constrained knapsack problem. The KS procedure is also related to the notion of *backbone* which was successfully applied to solve several binary optimization problems such as satisfiability (Zhang, 2004) and unconstrained binary quadratic programming (Wang et al., 2013). This is the first application of this idea to SUKP. Notice that given the particular feature of SUKP, our way of defining (and identifying) kernels remains unique compared to previous studies.

2.6. Direct perturbation procedure

The direct perturbation procedure aims to diversify the TS-KS process, by modifying the input local optimum S_l to generate a new starting solution for the next round of the TS-KS process. Specifically, the perturbation performs δ random *swap(q,p)* ($q \in \{0,1\}$, $p \in \{0,1\}$, and excluding $swap(q,p)$ with $q = p = 0$) operations to transform the input solution while ensuring the feasibility of the resulting solution, where δ is a parameter called *direct perturbation strength*. It is clear that larger δ values lead to more important changes of the input solution.

2.7. Non-kernel search procedure

When the TS and KS procedures (lines 9-19, Alg. 1) terminate, we employ a global diversification strategy to definitively drive the search to a faraway new region. To identify this new region, we refer to the kernel solution $S_k = \{y_1, \ldots, y_m\}$ (described in Section 2.5) and define its opposite solution $\bar{S_k} = \{x_1, \ldots, x_m\}$ such that $x_i = 1 - y_i$ ($i = 1, \ldots, m$). Then a feasible solution S is created from $\bar{S_k}$ and used as the input of the non-kernel search procedure. In order to obtain the feasible input solution S, we randomly select items from $\bar{S_k}$

and add them to S until the knapsack constraint is reached. The non-kernel search procedure follows the same search scheme (Algorithm 5) as TS and KS, but explores a different neighborhood $\bar{N}_k$ defined as follows. Specifically, during the non-kernel search, a swap operation is constrained to items that do not belong to the kernel S_k. In other words, items of S_k are never selected to become a part of a neighbor solution. As such, the non-kernel search has a strong diversification effect. The NKS procedure stops when the neighborhood becomes empty and the best solution found is used to initiate the next iteration of the whole KBTS algorithm.

Algorithm 5 Non-Kernel Search

Input: Input non-kernel solution $\bar{S_k}$, neighborhood $\bar{N_k}$.
Output: Best solution S_c found during non-kernel search.
$S \leftarrow Random(\bar{S_k})$ /* Generate a feasible solution from $\bar{S_k}$ */
$S_c \leftarrow S$ /*S_c records the best solution found during non-kernel search */
$Continue \leftarrow True$
while $Continue$ **do**
  $(Continue, S) \leftarrow$ Neighborhood_Search(S, $\bar{N_k}$, $Continue$)
  if $f(S) > f(S_c)$ **then**
    $S_c \leftarrow S$ /* Update the best solution found during non-kernel search */
  end if
end while
return S_c

2.8. Time complexity

We first consider the dynamic initialization procedure, which can be divided into two steps. The first step of updating *dynamic profit-ratio* can be achieved in $O(m^2n)$, and the second step of finding the non-selected item with the highest r_k^* value is bounded by $O(m^2)$, where m is the number of items and n is the number of elements. Thus the time complexity of the dynamic initialization procedure is $O(m^2n)$.

Now we evaluate one iteration of the main loop of the proposed algorithm. As shown in Algorithm 1, the tabu search procedure (TS), the kernel search procedure (KS) and the non-kernel search procedure (NKS) all adopt the Neighborhood_Search (NS) framework. Given the current solution $S =< A, \bar{A} >$ (see Section 2.4.1), the kernel solution S_k (see Section 2.5), and the non-kernel solution $\bar{S_k}$ (see Section 2.7), the corresponding complexity of one round of NS during the three procedures is $O([(m+|A|\times|\bar{A}|)]\times n)$, $O([(m-|S_k|)+(|A|-|S_k|)\times|\bar{A}|]\times n)$ and $O([|\bar{S_k}|+|A|\times(|\bar{S_k}|-|A|)]\times n)$. The complexity of the direct perturbation procedure is $O(1)$. Let R_{max} be the total maximum rounds of NS invoked by the TS, KS and NKS procedures. Then, the time complexity of one loop of KBTS is $O(m^2n \times R_{max})$.

Let I_{max} be the maximum number of the iterations of the KBTS algorithm (which is determined by the cut-off time t_{max}). Then, the overall time complexity of KBTS is $O(m^2n \times R_{max} \times I_{max})$. In Sections 3.3 and 4.4, we investigate the implications on the practical use of the above theoretical time complexity in terms of computational efficiency compared to existing SUKP algorithms.

2.9. Discussions

To highlight the novelties and contributions of the KBTS algorithm, we discuss below the main original features integrated in its search components.

First, the initialization procedure of Section 2.3 relies on an original *dynamic profit-ratio*. This strategy exploits the particular feature of SUKP that the elements of selected items can be reused regardless how many times they appear in the selected items of the current solution. The *dynamic profit-ratio* is thus a refined criterion compared to the *static profit-ratio* used in (Wei & Hao, 2019) and indeed favors the creation of high-quality initial solutions.

Second, the tabu search procedure of Section 2.4 has several special features that are different from other TS methods for SUKP (Lin et al., 2019; Wei & Hao, 2019). KBTS uses a parameter-free automatic tabu list strategy, while some parameters are required to control the tabu list and the tabu search termination in previous TS algorithms. Also, KBTS adopts an aspiration criterion to ensure that the best solution encountered is never overlooked, while no aspiration criterion is used in previous studies (Lin et al., 2019; Wei & Hao, 2019).

Third, although the general idea of *kernel* (or *backbone*) is known in the literature, we investigate for the first time the benefit of applying this idea to solve SUKP and propose a new way of identifying and using the kernel with the KBTS algorithm. Specifically, we extract the most frequent items from a set of high-quality solutions and use them to form a *kernel* solution (S_k). We additionally employ a parameter (*kernel coefficient*) to flexibly control the size of S_k within a proper range, which allows the kernel search procedure of Section 2.5 to intensively examine a given search region delimited by the kernel.

Fourth, the non-kernel search procedure of Section 2.7 relies on the opposite solution $\bar{S}_k$ of the kernel S_k. This is an original diversification strategy and has the advantage of diversifying the search in a guided manner. To our knowledge, such a strategy is not employed in the literature on SUKP.

Finally, as we demonstrate in the next section, the KBTS algorithm equipped with these innovative features is able to compete very favorably with the current best algorithms for SUKP in the literature.

3. Computational results and comparisons

This section is dedicated to an extensive evaluation of our KBTS algorithm and comparisons with state-of-the-art SUKP algorithms. We report computational results on two sets of 60 benchmark instances, available at `http://www.info.univ-angers.fr/pub/hao/SUKP_KBTS.html`.

3.1. Benchmark instances

Set I (30 instances): Introduced in (He et al., 2018), this set of instances have 85 to 500 items and elements with the following features. For each instance with m items and n elements, the items and elements are associated by a $m \times n$ binary relation matrix R, where $R_{ij} = 1$ indicates that item i includes element j. Each instance is further characterized by two parameters: α represents the

density of $R_{ij} = 1$ in the relation matrix R (i.e., $\alpha = (\sum_{i=1}^{m} \sum_{j=1}^{n} R_{ij})/(mn)$), β denotes the ratio of knapsack capacity C to the total weight of the elements (i.e., $\beta = C/\sum_{j=1}^{n} w_j$). Thus each SUKP instance can be designated as $m\_n\_\alpha\_\beta$. These instances are widely tested in the literature including (He et al., 2018; Lin et al., 2019; He & Wang, 2018; Ozsoydan & Baykasoğlu, 2018; Baykasoğlu et al., 2018; Ozsoydan, 2019; Feng et al., 2019a,b; Wei & Hao, 2019; Wu & He, 2020; Liu & He, 2019).

Set II (30 instances): Introduced in this work, this set of instances have the same characteristics as those of Set I, but are large in size with 585 to 1000 items and elements. Following (He et al., 2018), the profit and weight values of these instances are generated randomly in [1,500].

3.2. Experimental protocol and reference algorithms

Computing platform. Our KBTS algorithm is programmed in C++[1] and compiled with the g++ compiler with the -O3 option. To ensure a fair comparison, all the experiments mentioned in this work were performed on an Intel Xeon E5-2670 processor (2.5 GHz CPU and 2 GB RAM) running under the Linux operating system.

Parameter settings. The KBTS algorithm employs three parameters, whose descriptions and values are presented in Table 1. The effects and calibration of these parameters are presented in Section 4.1. The values of Table 1 can be considered to be the default setting and are used consistently to solve all 60 instances presented in Section 3.1 without any further fine-tuning.

Table 1: Parameters settings of KBTS.

Parameters	Section	Description	Value
γ_{max}	2.1	*local search depth*	3
ε	2.5	*kernel coefficient*	0.6
δ	2.6	*direct perturbation strength*	3

Reference algorithms. We adopt three recent state-of-the-art algorithms: hybrid jaya algorithm (DHJaya) (Wu & He, 2020), hybrid binary particle swarm optimization with tabu search (HBPSO/TS) (Lin et al., 2019) and iterated two-phase local search algorithm (I2PLS) (Wei & Hao, 2019). We also include the first binary artificial bee colony algorithm (BABC) (He et al., 2018) as a base reference. To ensure a fair comparison, we run the source codes of these algorithms (kindly provided by their authors) as well as our KBTS algorithm on our computing platform under the same stopping condition.

Stopping condition. Following (Wei & Hao, 2019), we run our KBTS algorithm and each reference algorithm to solve each of the 30 instances of Set I with a cut-off time of 500 seconds. For the 30 new large instances of Set II, the

[1]The code of our KBTS algorithm will be available at: `http://www.info.univ-angers.fr/pub/hao/SUKP_KBTS.html`.

cut-off time is set to 1000 seconds. Given the stochastic nature of the compared algorithms, each instance is independently solved by each algorithm 100 times with different random seeds.

3.3. Computational results and comparisons

Tables 2 and 3 present the detailed computational results[2] of the compared algorithms achieved on the two sets of benchmark instances. Column 1 gives the names of the tested instances while the asterisk (*) indicates the optimal value that are proved by CPLEX and reported in (Wei & Hao, 2019). The best objective value (f_{best}), the average objective value over 100 runs (f_{avg}), standard deviation over 100 runs (std) and the average run time (to reach the f_{best} value, denoted by t_{avg}) of each compared algorithm are reported in the remaining columns. In addition, the last row #Avg of Tables 2 and 3 indicates the average value of each column. Finally, dominating values of f_{best} and f_{avg} among the compared results are indicated in bold, and equal best values are shown in italic.

From the results of Table 2 on the instances of Set I, we observe that our KBTS algorithm is very competitive compared to the reference algorithms in terms of f_{best}, f_{avg} and std. Also, KBTS has a better average performance and very small standard deviations, indicating its high robustness. The high competitiveness of our KBTS algorithm becomes even more evident when we check the results of Table 3 for the 30 large instances of Set II. Indeed, KBTS dominates all the reference algorithms in all performance indicators. Moreover, KBTS requires less computation times to attain better solutions with small standard deviations, indicating its high computational efficiency and robustness.

Fig. 2 additionally shows a graphical representation of the comparative results of the five competing algorithms on the two sets of instances in terms of the best objective values, the average objective values and the standard deviations. The X-axis in each sub-figure indicates the 30 instances of each set and the Y-axis gives the f_{best}, f_{avg} and std values of the compared algorithms. The plots of Fig. 2 clearly indicate the dominance of our KBTS algorithm over the reference algorithms and its particular advantage on the set of large instances.

[2]Our solution certificates are available at: `http://www.info.univ-angers.fr/pub/hao/SUKP_KBTS.html`.

Table 2: Computational results and comparison of the KBTS algorithm with the reference algorithms on the SUKP instances of Set I.

Instance	BABC				DHJaya				HBPSO/TS				I2PLS (Best_Known)				KBTS			
	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	51.102	*13283*	*13283*	0	9.477	*13283*	*13283*	0	0.098	*13283*	*13283*	0	3.094	*13283*	*13283*	0	4.082
100_85_0.15_0.85*	*12479*	*12479*	0	24.032	*12479*	*12479*	0	24.414	*12479*	12403.15	98.97	101.122	*12479*	12335.13	98.78	103.757	*12479*	*12479*	0	42.992
200_185_0.10_0.75	13402	13260.16	38.98	253.693	*13521*	13498.22	26.10	258.213	*13521*	*13521*	0	0.490	*13521*	*13521*	0	71.984	*13521*	*13521*	0	6.988
200_185_0.15_0.85	*14215*	14026.18	151.55	241.932	*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
300_285_0.10_0.75	10572	10466.45	61.94	315.240	11385	11167.77	129.98	174.335	*11563*	*11563*	0	38.355	*11563*	11562.02	3.94	181.248	*11563*	*11563*	0	28.841
300_285_0.15_0.85	12245	12019.28	85.76	226.818	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
400_385_0.10_0.75	11021	10608.91	138.07	293.560	*11484*	11325.88	38.65	229.370	*11484*	*11484*	0	10.870	*11484*	*11484*	0	31.801	*11484*	*11484*	0	0.296
400_385_0.15_0.85	9649	9503.65	94.69	270.813	10710	10293.96	173.85	241.068	*11209*	*11209*	0	16.478	*11209*	11157.26	87.29	141.525	*11209*	*11209*	0	72.020
500_485_0.10_0.75	10927	10628.31	70.31	486.210	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
500_485_0.15_0.85	9306	9014.01	64.06	482.740	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
100_100_0.10_0.75*	*14044*	14040.87	11.51	169.848	*14044*	*14044*	0	1.374	*14044*	*14044*	0	0.518	*14044*	*14044*	0	38.245	*14044*	*14044*	0	0.023
100_100_0.15_0.85*	*13508*	*13508*	0	6.795	*13508*	*13508*	0	1.572	*13508*	*13508*	0	2.923	*13508*	13451.50	126.49	70.587	*13508*	*13508*	0	33.403
200_200_0.10_0.75	12350	11953.11	97.57	183.130	*12522*	12480.62	65.05	207.667	*12522*	*12522*	0	0.8125	*12522*	*12522*	0	54.780	*12522*	*12522*	0	48.206
200_200_0.15_0.85	11929	11695.21	78.33	147.930	*12317*	12217.81	93.361	229.824	*12317*	*12317*	0	0.950	*12317*	12280.07	57.77	238.348	*12317*	*12317*	0	72.495
300_300_0.10_0.75	12304	12202.80	67.81	202.515	12736	12676.78	35.20	241.774	*12817*	12806.44	15.39	29.074	*12817*	*12817*	0	66.403	*12817*	*12817*	0	74.247
300_300_0.15_0.85	10857	10383.64	75.79	113.380	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
400_400_0.10_0.75	10869	10591.65	105.83	298.970	11569	11301.56	74.88	322.143	*11665*	11484.20	72.95	45.025	*11665*	*11665*	0	18.733	*11665*	*11665*	0	64.126
400_400_0.15_0.85	10048	9602.13	142.77	386.555	10927	10721.45	221.38	77.037	*11325*	*11325*	0	5.902	*11325*	*11325*	0	76.000	*11325*	*11325*	0	17.591
500_500_0.10_0.75	10755	10522.56	70.17	194.490	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
500_500_0.15_0.85	9601	9334.52	40.59	135.130	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
85_100_0.10_0.75*	*12045*	11995.12	53.15	206.570	*12045*	*12045*	0	17.199	*12045*	*12045*	0	0.056	*12045*	*12045*	0	2.798	*12045*	*12045*	0	0.075
85_100_0.15_0.85*	*12369*	*12369*	0	0.531	*12369*	*12369*	0	0.342	*12369*	*12369*	0	0.088	*12369*	12315.53	62.60	17.470	*12369*	*12369*	0	10.175
185_200_0.10_0.75	13647	13179.14	100.78	202.560	*13696*	13667.63	26.56	244.205	*13696*	*13696*	0	0.489	*13696*	13695.60	3.68	124.136	*13696*	*13696*	0	5.851
185_200_0.15_0.85	10926	10749.46	97.24	259.050	*11298*	*11298*	0	38.439	*11298*	*11298*	0	0.486	*11298*	11276.17	83.78	139.865	*11298*	*11298*	0	6.373
285_300_0.10_0.75	11374	11143.69	76.90	426.680	*11568*	11563.80	10.41	203.874	*11568*	*11568*	0	13.630	*11568*	*11568*	0	25.128	*11568*	*11568*	0	30.618
285_300_0.15_0.85	10822	10396.60	128.63	192.575	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
385_400_0.10_0.75	10110	9926.18	87.43	203.870	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
385_400_0.15_0.85	9659	9444.34	46.40	177.910	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
485_500_0.10_0.75	10835	10789.57	27.29	299.260	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
485_500_0.15_0.85	9380	9258.82	58.72	49.170	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
#Avg	11484.37	11279.18	69.08	216.769	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

Table 3: Computational results and comparison of the KBTS algorithm with the reference algorithms on the SUKP instances of Set II.

Instance	BABC				DHJaya				HBPSO/TS				I2PLS				KBTS			
	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	9098	9026.05	34.87	498.591	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
600_585_0.15_0.85	8736	8540.46	20.51	172.475	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
700_685_0.10_0.75	9311	9176.28	46.93	363.381	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
700_685_0.15_0.85	8671	8397.36	87.65	302.624	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
800_785_0.10_0.75	9275	9192.36	20.27	253.268	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
800_785_0.15_0.85	8447	8366.50	71.97	254.293	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
900_885_0.10_0.75	8953	8837.18	103.15	471.428	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
900_885_0.15_0.85	8072	7881.17	88.49	228.388	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
1000_985_0.10_0.75	9276	9254.19	27.89	640.529	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
1000_985_0.15_0.85	8133	8099.10	25.37	648.215	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
600_600_0.10_0.75	10207	9939.38	47.52	66.660	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
600_600_0.15_0.85	8621	8361.77	101.30	455.481	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
700_700_0.10_0.75	9078	9056.52	21.89	224.370	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
700_700_0.15_0.85	8614	8290.22	77.62	126.818	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
800_800_0.10_0.75	9517	9305.40	56.76	418.476	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
800_800_0.15_0.85	8444	8163.77	132.71	376.695	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
900_900_0.10_0.75	9290	9272.99	14.56	460.026	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
900_900_0.15_0.85	8118	8114.48	9.20	150.984	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
1000_1000_0.10_0.75	9030	8891.34	39.01	657.972	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
1000_1000_0.15_0.85	7867	7627.80	44.88	635.003	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
585_600_0.10_0.75	9768	9677.80	81.90	535.874	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
585_600_0.15_0.85	8689	8623.79	28.52	461.850	9031	8944.22	61.72	616.631	*9256*	*9256*	0	103.637	*9256*	*9256*	0	264.876	*9256*	*9256*	0	84.359
685_700_0.10_0.75	9796	9627.40	73.18	248.733	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
685_700_0.15_0.85	8453	8424.87	4.83	958.748	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
785_800_0.10_0.75	8765	8658.45	54.33	869.031	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
785_800_0.15_0.85	8249	8021.86	117.07	577.037	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
885_900_0.10_0.75	8938	8897.58	30.23	587.200	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
885_900_0.15_0.85	7610	7518.04	50.51	869.729	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
985_1000_0.10_0.75	8914	8741.25	101.76	739.861	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
985_1000_0.15_0.85	8071	8066.53	15.17	486.522	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
#Avg	8800.37	8668.40	54.33	458.009	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

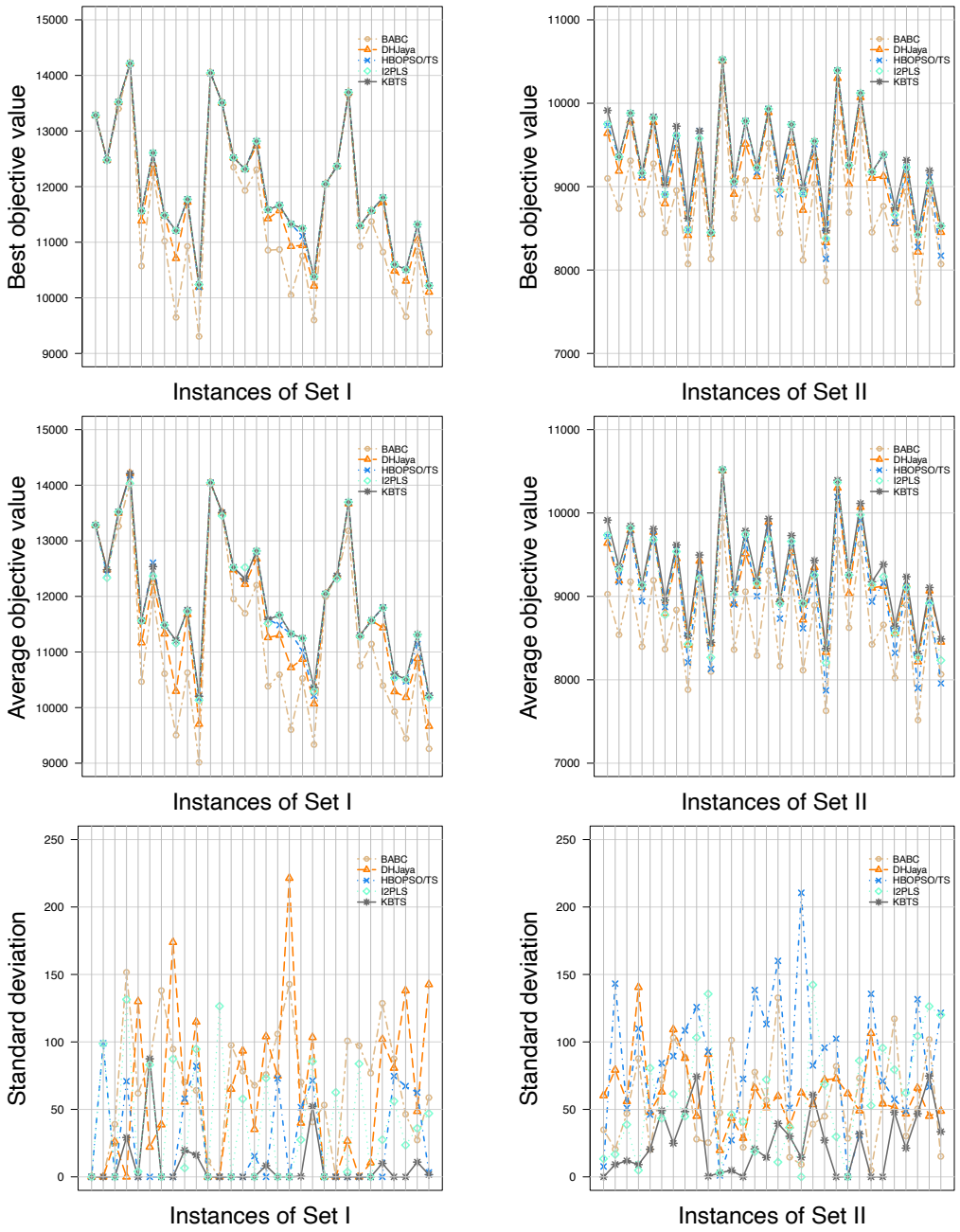

Fig. 2. Best objective values, average objective values and standard deviations of BABC, DHJaya, HBPSO/TS, I2PLS and KBTS on the 30 instances of Set I (left) and the 30 instances of Set II (right).

Finally, Table 4 summarizes the comparative results between the KBTS

Table 4: Summarized comparisons of the KBTS algorithm against each reference algorithm with the *p-values* of the Wilcoxon signed-rank test over the two sets of benchmark instances.

Algorithm pair	Instance set	Indicator	#Wins	#Ties	#Losses	*p-value*
KBTS vs. BABC	Set I (30)	f_{best}	23	7	0	2.70e-5
		f_{avg}	26	4	0	8.30e-6
	Set II (30)	f_{best}	30	0	0	1.73e-6
		f_{avg}	30	0	0	1.73e-6
KBTS vs. DHJaya	Set I (30)	f_{best}	16	14	0	4.38e-4
		f_{avg}	22	7	1	3.53e-5
	Set II (30)	f_{best}	30	0	0	1.73e-6
		f_{avg}	30	0	0	1.73e-6
KBTS vs. HBPSO/TS	Set I (30)	f_{best}	2	28	0	1.80e-1
		f_{avg}	12	15	3	7.60e-3
	Set II (30)	f_{best}	18	12	0	8.85e-5
		f_{avg}	29	1	0	2.56e-6
KBTS vs. I2PLS	Set I (30)	f_{best}	0	30	0	NA
		f_{avg}	20	10	0	1.51e-3
	Set II (30)	f_{best}	13	17	0	1.32e-4
		f_{avg}	29	1	0	2.56e-6

algorithm and each reference algorithm. This table focuses on the f_{best} and f_{avg} indicators and shows the number of instances achieved by KBTS to obtain a better, an equal or a worse result (#Wins, #Ties and #Losses) compared to each reference algorithm. To verify the statistical significance of the comparisons of KBTS against the reference algorithms, the *p-values* from the non-parametric Wilcoxon signed-rank test are shown in the last column. And a *p-value* less than 0.05 implies a significant difference between KBTS and its competitor, while 'NA' means that the two sets of compared results are exactly the same. This summarized comparison clearly confirms the high performance of our KBTS algorithm. Indeed, for a majority of the tested instances, KBTS always reports better or equal results in terms of f_{best} and f_{avg}. Such a performance was never attained by any reference algorithm.

4. Analysis

In this section, we present an analysis of the parameters used in the proposed algorithm and the kernel based components.

4.1. Analysis of parameters

The proposed KBTS algorithm requires three parameters: *kernel coefficient* ε, *local search depth* γ_{max} and *direct perturbation strength* δ. We first carry out a factorial experiment (Montgomery, D. C. , 2017) to gain insights into the effect of parameters on the algorithm performance and then perform a one-at-a-time

Table 5: Parameter levels for the 2-level full factorial experiment.

	Low level	High level
kernel coefficient ε	0.3	0.6
local search depth γ_{max}	3	6
direct perturbation strength δ	3	6

sensitivity analysis (Hamby, 1994) to calibrate the parameters. For these experiments, we select eight representative instances from Set II: 785_800_0.15_0.85, 800_785_0.15_0.85, 800_800_0.15_0.85, 885_900_0.15_0.85, 900_885_0.15_0.85, 985_1000 _0.10_0.75, 1000_985_0.10_0.75 and 1000_1000_0.10_0.75. These instances are difficult since the results reported by different algorithms (see Table 3) show large standard deviations.

We employ a 2-level full factorial experiment to observe the interaction effects between the parameters. The levels of the three parameters are shown in Table 5. For this experiment, each instance was independently solved 20 times with different combinations of parameters. Then we consider the average value of the best objective values (f_{best}) obtained on the eight instances for each parameter combination. We verify the normality of data distributions and the variance homogeneity. We show the main effects of the parameters in Fig. 3 and the analysis of the variances in Table 6.

From Fig. 3, we can observe that the effects of the parameter *kernel coefficient* and *local search depth* are positive, while the effect of direct perturbation strength is negative. The *p-values* (< 0.05) in columns 2-3 of Table 6 indicate that the performance of the algorithm is sensitive to the setting of *kernel coefficient* and *local search depth*. Moreover, it makes sense to check the interaction effects between the parameters. From Table 6, we can observe that the *p-values* of the last four columns are all greater than 0.05, which indicates that the interaction effects among the parameters are not statistically significant.

Now we perform a one-at-a-time sensitivity analysis to determine a suitable value for each parameter. Based on a reasonable range of parameter values: $\varepsilon \in \{0.1, 0.2, ..., 1\}$, $\gamma_{max} \in \{1, 2, ..., 10\}$ and $\delta \in \{1, 2, ..., 10\}$, we test the values of each parameter independently while keeping the other parameters fixed to the values of Table 1. For this, we run the algorithm with each parameter setting 30 times to solve each instance. Fig. 4 shows the average of the best objective values (f_{best}) attained by KBTS with different parameter settings. The X-axis indicates the ranges of the three parameters, i.e., 1 to 10 for γ_{max} and δ, 0.1 to 1 for ε. From Fig. 4, we observe that KBTS reaches its best performance with $\varepsilon = 0.6$, $\gamma_{max} = 3$ and $\delta = 3$. These values are thus used to define the default parameter setting shown in Table 1 of Section 3.2.

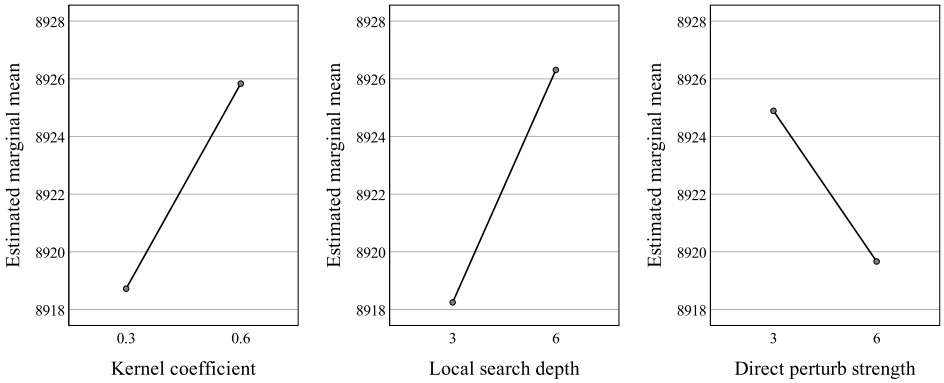

Fig. 3. Effects of the three parameters on the performance of the KBTS algorithm.

Table 6: *p-values* for the analysis of variances with the significance level 0.05.

Source of variation	ε	γ_{max}	δ	ε * γ_{max}	ε * δ	γ_{max} * δ	ε * γ_{max} * δ
p-value	3.70e-2	1.80e-2	1.25e-1	3.90e-1	1.47e-1	1.92e-1	8.41e-1

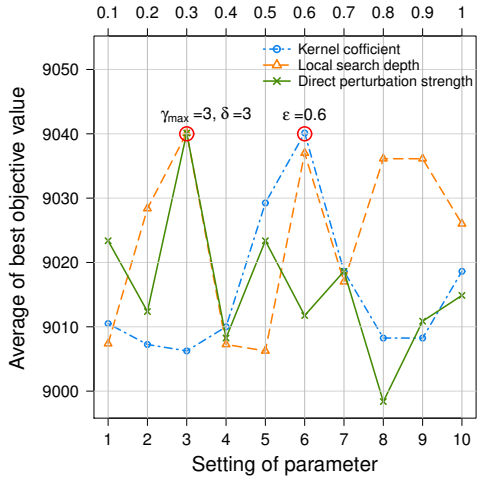

Fig. 4. Average of the best objective values (f_{best}) corresponding to different parameter settings obtained by the one-at-a-time sensitivity analysis.

4.2. Impact of kernel search and non-kernel search

The proposed KBTS algorithm relies on the notion of kernel and the associated kernel search and non-kernel search procedures. To assess the usefulness of

these components, we create a KBTS variant (denoted by KBTS$^-$) by disabling the kernel search procedure (i.e., removing line 11 in Alg. 1) and replacing the non-kernel search procedure with a random strategy (i.e., we generate randomly a feasible solution S of line 23 in Alg. 1). We run KBTS and KBTS$^-$ 30 times according to the experimental protocol given in Section 3.2 to solve each instance of Set II and report the results in Table 7. In this table, we show the f_{best}, f_{avg} and std values. The row #Avg indicates the average value of each column and the row #Best shows the number of instances for which an algorithm achieves the best results between the two set of results.

The results show that compared to KBTS, the KBTS$^-$ variant obtains worse f_{best} values for 7 instances, and worse f_{avg} values for 5 instances, leading to worse #Avg values of these performance indicators. Table 7 also indicates that KBTS$^-$ deteriorates the results of KBTS for the most difficult instances (with 785 to 1000 items and elements), which reveals that the kernel search procedure is particularly useful for solving difficult instances. Furthermore, the Wilcoxon signed-rank tests in terms of f_{best} (p-$value < 0.05$) confirm that the performance differences between KBTS and KBTS$^-$ are statistically significant.

4.3. Distribution of high-quality solutions and rationale of kernel search

To understand why the notion of kernel is pertinent, we present a study on distributions of items in high-quality solutions. This study is based on a selection of four representative instances: 500_485_0.15_0.85, 500_500_0.15_0.85, 1000_1000_0.10_0.75, 1000_1000_0.15_0.85. For each instance, we run KBTS 30 times to obtain 30 high-quality solutions and then extract frequency statistics of selected items in these solutions, as shown in Fig. 5. The X-axis in each sub-figure indicates the number of selected items and the Y-axis refers to the frequency that one item appears in these solutions. We also present the number of items corresponding to each frequency on the right side of the Y-axis and the bottom value in this column corresponds to the number of items with a frequency of 0. Since this bottom value is much larger than the other values corresponding to the frequencies in the range $\{1, ..., 30\}$, we don't draw its corresponding plot for the convenience of observation.

From Fig. 5, we observe that the frequency of most items being selected in a solution is polarized, that is, these items are either selected many times or are rarely selected. In particular, almost 90% of the items in each of these four instances never belong to a high-quality solution. This experiment thus indicates that high-quality solutions often contain several identical items (which form a kernel), providing a supporting argument for the usefulness of the kernel based components of the KBTS algorithm.

Table 7: Comparison between KBTS (with the kernel components) and KBTS$^-$ (without the kernel components) on the instances of Set II.

Instance/Setting	KBTS			KBTS$^-$		
	f_{best}	f_{avg}	std	f_{best}	f_{avg}	std
600_585_0.10_0.75	9914	9914	0	9914	9800.70	77.56
600_585_0.15_0.85	9357	9353.47	11.29	9357	9356.40	3.23
700_685_0.10_0.75	9881	9845	12	9881	9851.47	17.36
700_685_0.15_0.85	9163	9137.80	8.40	9163	9138.73	9.52
800_785_0.10_0.75	9837	9810.80	16.56	9829	9806.57	17.10
800_785_0.15_0.85	9024	8944	43.36	9024	8935.07	45.08
900_885_0.10_0.75	9725	9614.80	20.46	9725	9614.80	20.46
900_885_0.15_0.85	8620	8534.57	54.15	8588	8541.73	54.39
1000_985_0.10_0.75	9668	9512.13	74.70	9668	9477.40	56.68
1000_985_0.15_0.85	8448	8448	0	8448	8448	0
600_600_0.10_0.75	10524	10521.60	2.94	10524	10521.60	2.94
600_600_0.15_0.75	9062	9061.07	5.03	9062	9060.73	6.82
700_700_0.10_0.75	9786	9786	0	9786	9786	0
700_700_0.15_0.85	9229	9185.60	19.51	9177	9177	0
800_800_0.10_0.75	9932	9932	0	9932	9932	0
800_800_0.15_0.85	9101	8935.83	40.92	9101	8928.77	39.09
900_900_0.10_0.75	9745	9731.40	29.25	9745	9741.03	16.24
900_900_0.15_0.85	8990	8920.93	18.46	8916	8916	0
1000_1000_0.10_0.75	9544	9424	55.68	9544	9424.37	51.06
1000_1000_0.15_0.85	8474	8379.33	24.19	8438	8374.33	20.79
585_600_0.10_0.75	10393	10393	0	10393	10393	0
585_600_0.15_0.85	9256	9256	0	9256	9256	0
685_700_0.10_0.75	10121	10112.80	35.87	10121	10121	0
685_700_0.15_0.85	9176	9176	0	9176	9176	0
785_800_0.10_0.75	9384	9384	0	9384	9384	0
785_800_0.15_0.85	8746	8650.43	48.04	8663	8645.60	27.77
885_900_0.10_0.75	9318	9239.47	26.88	9318	9233.57	17.29
885_900_0.15_0.85	8425	8312.43	47.17	8425	8319.97	46.16
985_1000_0.10_0.75	9193	9086.07	77.58	9186	9083.90	69.38
985_1000_0.15_0.85	8528	8497.93	33.15	8528	8484.83	36.00
#Avg	**9352.13**	**9303.35**	23.52	9342.40	9297.69	21.16
#Best	**30**	**22**	-	23	17	-
p-value	-	-	-	1.80e-2	2.31e-1	-

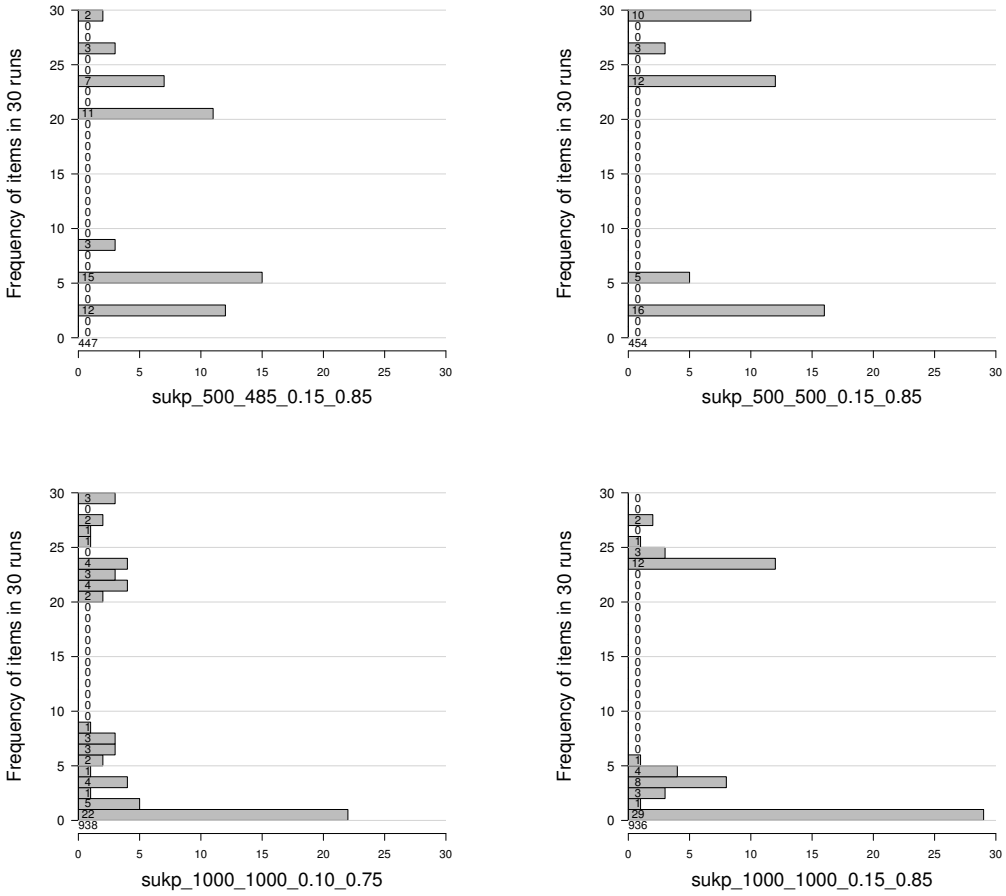

Fig. 5. Distributions of high-quality solutions corresponding to different item frequencies.

4.4. Time-to-target analysis

To further asses the computational efficiency of the proposed KBTS algorithm with respect to the reference algorithms (BABC, DHJaya, HBPSO/TS, I2PLS, and KBTS), we present a time-to-target (TTT) analysis (Aiex et al., 2007; Ribeiro et al., 2012). Basically, TTT shows the computation time required by an algorithm to attain a given target objective value. This analysis is based on four representative instances of Set II, i.e., 585_600_0.10_0.75, 600_600_0.15_0.85, 800_785_0.15_0.85, 1000_985_0.10_0.75. For each instance, we set the target value to be a value, which can be reached by all the compared algorithms (10000, 8800, 8700 and 9000, respectively) and record the time (over 100 runs) of each algorithm to reach a solution with an objective value at least as good as the given target value. The time-to-target plots are shown in Fig. 6, where the time required to achieve the target value and the corresponding cumulative probability

are displayed on the X-axis and Y-axis, respectively.

From Fig. 6, we observe that our KBTS algorithm has a very high computational efficiency, surpassing all the reference algorithms according to the cumulative probability. The lines of KBTS strictly runs above the lines of the reference algorithms, revealing that our algorithm has always a higher probability to reach the given target value.

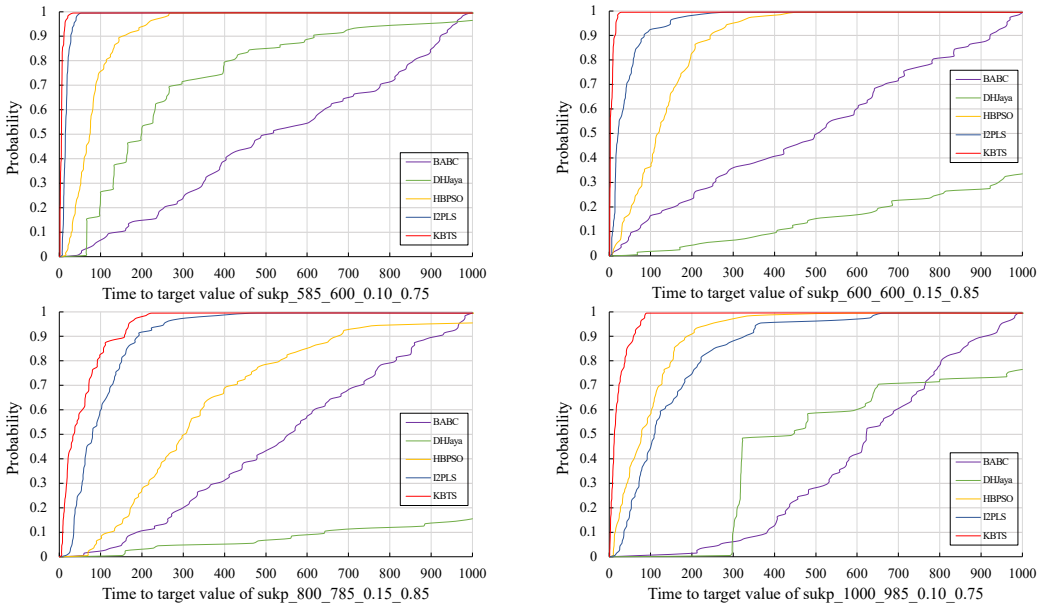

Fig. 6. Time-to-target plots of the compared algorithms on four SUKP instances.

5. Conclusions

The Set-union Knapsack Problem (SUKP) is a relevant model for decision making and intelligent systems. Given its intrinsic difficulty (NP-hard), heuristic algorithms are useful to find high-quality solutions in a reasonable time frame. We presented the kernel based tabu search algorithm, which combines for the first time the notion of kernel with the powerful tabu search method.

Our computational study performed on two sets of 60 benchmark instances indicated that the proposed algorithm dominates the current best SUKP algorithms in the literature in terms of solution quality, robustness and computation time. This dominance was particularly evidenced on large and difficult benchmark instances with at least 500 items and elements. Compared to the existing SUKP algorithms, the proposed algorithm requires only three parameters, making it more suitable to use in practice. Given that SUKP has a number of interesting applications, the proposed algorithm provides a valuable tool for solving these real world problems. The availability of the source code of our algorithm and its high computational efficiency certainly facilitate such applications.

For future work, we identify three perspectives. First, one can investigate other ways to obtain the kernel solution, e.g., by using frequent pattern mining technology. Second, SUKP is a constrained problem, it would be interesting to investigate mixed search strategies that explore both feasible and infeasible solutions. Third, solution-based tabu search has shown good performances on other knapsack problems (e.g., Lai et al. (2018a)). Studying this approach constitutes a promising direction for better solving SUKP. Finally, the proposed algorithm or its variants can be embedded into population based frameworks (e.g., memetic computing methods) to obtain more powerful algorithms.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships 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 Prof. Yichao He, Dr. Congcong Wu, Dr. Geng Lin and their co-authors for sharing the codes of their algorithms (BABC (He et al., 2018), DHJaya (Wu & He, 2020), HBPSO/TS (Lin et al., 2019)). Support from the China Scholarship Council (Grant 201706290016) for the first author is also acknowledged.

References

Aiex, R. M., Resende, M. G., & Ribeiro, C. C. (2007). TTT plots: a perl program to create time-to-target plots. *Optimization Letters*, 1(4):355–366.

Amiri, A. (2020). A Lagrangean based solution algorithm for the knapsack problem with setups. *Expert Systems with Applications*, 143, 113077.

Arulselvan, A. (2014). A note on the set union knapsack problem. *Discrete Applied Mathematics*, 169, 214–218.

Baykasoğlu, A., Ozsoydan, F.B., & Senol, M. (2018). Weighted superposition attraction algorithm for binary optimization problems. *Operational Research*, 1–27. https://doi.org/10.1007/s12351-018-0427-9.

Dahmani, I., Hifi, M., Saadi, T., & Yousef, L. (2020). A swarm optimization-based search algorithm for the quadratic knapsack problem with conflict Graphs. *Expert Systems with Applications*, 148, 113224.

Denysiuk, R., Gaspar-Cunha, A., & Delbem, A. C. (2019). Neuroevolution for solving multiobjective knapsack problems. *Expert Systems with Applications*, 116, 65-77.

Díaz, J.A., Luna, D.E., Camacho-Vallejo, J.F., & Casas-Ramírez, M.S. (2017). GRASP and hybrid GRASP-Tabu heuristics to solve a maximal covering location problem with customer preference ordering. *Expert Systems with Applications*, 821, 67–76.

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

Feng, Y.H., Yi, J.H., & Wang, G.G. (2019b). Enhanced Moth Search Algorithm for the Set-Union Knapsack Problems. *IEEE Access*, 7, 173774–173785.

Glover, F., & Kochenberger, G.A. (1996). Critical event tabu search for multidimensional knapsack problems. In Osman I.H., Kelly J.P. (eds) *Meta-Heuristics*, (pp. 407-427). Springer, Boston, MA.

Glover, F., & Laguna, M. (1997). Tabu Search. Springer Science+Business Media New York.

Goldschmidt, O., Nehme, D., & Yu, G. (1994). Note: On the set-union knapsack problem. *Naval Research Logistics*, 41(6), 833–842.

Hamby, D.M. (1994). A review of techniques for parameter sensitivity analysis of environmental models. *Environmental Monitoring and Assessment*, 32(2), 135–154.

He, Y.C., & Wang, X.Z. (2018). Group theory-based optimization algorithm for solving knapsack problems. *Knowledge-Based Systems*, 104445.

He, Y.C., Xie, H.R., Wong, T.L., & Wang, X.Z. (2018). A novel binary artificial bee colony algorithm for the set-union knapsack problem. *Future Generation Computer Systems*, 78, 77–86.

Kellerer, H., Pferschy, U., & Pisinger, D. (2004). Knapsack problems. Springer.

Lai, X.J, Hao, J.K., & Yue, D. (2018a). Two-stage solution-based tabu search for the multidemand multidimensional knapsack problem. *European Journal of Operational Research*, 274(1), 35–48.

Lai, X.J, Hao, J.K., Glover, F., & Lü, Z.P. (2018b). A two-phase tabu-evolutionary algorithm for the 0-1 multidimensional knapsack problem. *Information Sciences*, 436, 282–301.

Lai, X.J, Hao, J.K., & Glover, F. (2020). A study of two evolutionary/tabu search approaches for the generalized max-mean dispersion problem. *Expert Systems with Applications*, 139, 112856.

Lin, G., Guan, J., Li, Z.Y., & Feng, H.B. (2019). A hybrid binary particle swarm optimization with tabu search for the set-union knapsack problem. *Expert Systems with Applications*, 135, 201–211.

Lister, W., Laycock, R.G., & Day, A.M. (2010). A key-pose caching system for rendering an animated crowd in real-time. *Computer Graphics Forum*, 29(8), 2304–2312.

Liu, X.J., & He, Y.C. (2019). Estimation of distribution algorithm based on Lévy flight for solving the set-union knapsack problem. *IEEE Access*, 7, 132217–132227.

Montgomery, D. C. (2017). Design and analysis of experiments. *John Wiley & Sons*.

Navathe, S., Ceri, S., Wiederhold, G., & Dou, J.L. (1984). Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems*, 9(4), 680–710.

Ozsoydan, F.B. (2019). Artificial search agents with cognitive intelligence for binary optimization problems. *Computers and Industrial Engineering*, 136, 18–30.

Ozsoydan, F.B., & Baykasoğlu, A. (2019). A swarm intelligence-based algorithm for the set-union knapsack problem. *Future Generation Computer Systems*, 93, 560–569.

Qin, J., Xu, X.H., Wu, Q.H., & Cheng, T.C.E. (2016). Hybridization of tabu search with feasible and infeasible local searches for the quadratic multiple knapsack problem. *Computers and Operations Research*, 66, 199–214.

Ribeiro, C. C., Rosseti, I., & Vallejos, R. (2012). Exploiting run time distributions to compare sequential and parallel stochastic local search algorithms. *Journal of Global Optimization*, 54(2):405–429.

Schneier, B. (1996). Applied cryptography: protocols, algorithms, and source code in C. Second edition. *John Wiley & Sons*.

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

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

Vasquez, M., & Hao, J.K. (2001). A "logic-constrained" knapsack formulation and a tabu algorithm for the daily photograph scheduling of an earth observation satellite. *Computational Optimization and Applications*, 20(2), 137–157.

Wang, Y., Lü, Z.P., Glover, F., & Hao, J.K. (2013). Backbone guided tabu search for solving the UBQP problem. *Journal of Heuristics*, 19(4), 679–695.

Wei, Z.Q., & Hao, J.K. (2019). Iterated two-phase local search for the Set-Union Knapsack Problem. *Future Generation Computer Systems*, 101, 1005–1017.

Wu, C.C., & He, Y.C. (2020). Solving the set-union knapsack problem by a novel hybrid Jaya algorithm. *Soft Computing*, 24(3), 1883–1902.

Zhang, W.X. (2004). Configuration landscape analysis and backbone guided local search. Part I: Satisfiability and maximum satisfiability. *Artificial Intelligence*, 158(1), 1–26.