Expert Systems With Applications 135 (2019) 201–211

Contents lists available at ScienceDirect

Expert Systems With Applications

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

A hybrid binary particle swarm optimization with tabu search for the set-union knapsack problem

Geng Lin[a,b,*], Jian Guan[c], Zuoyong Li[d], Huibin Feng[d]

[a] Collaborative Innovation Center of IoT Industrialization and Intelligent Production, Minjiang University, Fuzhou, 350121, China
[b] College of Mathematics and Data Science, Minjiang University, Fuzhou 350121, China
[c] Modern Educational Technology Center, Minjiang University, Fuzhou 350121, China
[d] Fujian Provincial key Laboratory of Information Processing and Intelligent Control, Minjiang University, Fuzhou 350121, China

ARTICLE INFO

Article history:
Received 26 October 2018
Revised 4 June 2019
Accepted 5 June 2019
Available online 6 June 2019

Keywords:
Set-union knapsack problem
Particle swarm optimization
Local search
Heuristic
Binary programming

ABSTRACT

The set-union knapsack problem (SUKP) is a generalization of the standard 0–1 knapsack problem. It is NP-hard, and has several industrial applications. Several approximation and heuristic approaches have been previously presented for solving the SUKP. However, the solution quality still needs to be enhanced. This work develops a hybrid binary particle swarm optimization with tabu search (HBPSO/TS) to solve the SUKP. First, an adaptive penalty function is utilized to evaluate the quality of solutions during the search. This method endeavours to explore the boundary of the feasible solution space. Next, based on the characteristics of the SUKP, a novel position updating procedure is designed. The newly generated solutions obtain the good structures of previously found solutions. Then, a tabu based mutation procedure is introduced to lead the search to enter into new hopeful regions. Finally, we design a tabu search procedure to improve the exploitation ability. Furthermore, a gain updating strategy is employed to reduce the solution time. The HBPSO/TS is tested on three sets of 30 benchmark instances, and comparisons with current state-of-the-art algorithms are performed. Experimental results show that HBPSO/TS performs much better than the other algorithms in terms of solution quality. Moreover, HBPSO/TS improves new best results at 28 out of the 30 instances. The impact of the main parts of the HBPSO/TS is also experimentally investigated.

1. Introduction

The set-union knapsack problem (SUKP) (Arulselvan, 2014; Goldschmidt, Nehme, & Yu, 1994) is a generalization of the well-known 0–1 knapsack problem (Pisinger, 1995), in which the total weight of the item set is defined by the sum of the weight of elements in the item set. We define a set $U = \{1, 2, \dots, n\}$ of elements with weight w_j for each element $j \in U$, and a set $S = \{1, 2, \dots, m\}$ of items with profit p_i for each item $i \in S$. Each item $i \in S$ corresponds to a subset $U_i \subseteq U$. Given a set $A \subseteq S$, the profit sum and the weight sum of A can be calculated by:

$$P(A) = \sum_{i \in A} p_i, \tag{1}$$

and

$$W(A) = \sum_{j \in \bigcup_{i \in A} U_i} w_j, \tag{2}$$

respectively. The SUKP is to find a subset of S that maximizes the profit sum where the weight sum cannot be exceed a given capacity C. If $U_i \bigcap U_{i'} = \emptyset$ for $i, i' = 1, \dots, m, i \neq i'$, the SUKP becomes the standard 0–1 knapsack problem.

Let $x = (x_1, \dots, x_m)^T \in \{0, 1\}^m$ be a solution of the SUKP. $x_i = 1$ indicates that item i is selected, and $x_i = 0$ otherwise. Let $A_x = \{i \in S : x_i = 1\}$. It is obvious that the 0–1 vector x maps to the subset $A_x \in S$ one to one. Let $\Lambda(x)$ be the set of selected elements by A_x (or solution x):

$$\Lambda(x) = \bigcup_{i \in A_x} U_i. \tag{3}$$

The weight sum of A_x can be calculated by $W(A_x) = \sum_{j \in \Lambda(x)} w_j$. Subsequently, SUKP can be formulated as follows by constrained 0–1 programming (He, Xie, Wong, & Wang, 2018):

$$\begin{cases} \max & f(x) = \sum_{i=1}^{m} p_i x_i, \\ s.t. & \sum_{j \in \Lambda(x)} w_j \le C, \\ & x \in \{0, 1\}^m. \end{cases}$$

The SUKP has been shown to be NP-hard (Goldschmidt et al., 1994), and has been used in various domains including database

* Corresponding author.
E-mail addresses: lingeng413@163.com (G. Lin), 155295225@qq.com (J. Guan), 67580180@qq.com (Z. Li), 35426918@qq.com (H. Feng).

https://doi.org/10.1016/j.eswa.2019.06.007
0957-4174/

partitioning (Navathe, Ceri, Wiederhold, & Dou, 1984), financial decision making (Kellerer, Pferschy, & Pisinger, 2004), flexible manufacturing (Tang & Denardo, 1988), and smart cities (Tu & Xiao, 2016), etc. It has been also used in practical applications such as the key-pose caching problem (Lister, Laycock, & Day, 2010). In the key-pose caching problem, for each frame, there are n characters which will be rendered whereby each character interpolates a set of m key poses. For a character, its key-poses remain valid for a period of time, and can be shared with other characters. The key-pose caching problem maximizes the number of rendered characters by populating a cache of locations with skinned key-poses. Let the set of elements U be equal to the set of all key-poses needed by the crowd in a given frame, and the weight of each key-pose is set as 1. Each item refers to a crowd member, and its profit is equal to the number of occurrences of the key-pose pair. Then, the key-pose caching problem can be formulated as the SUKP.

Let I_j be the set of items that contain element j. More formally, $I_j = \{i \in S : j \in U_i\}$. The frequency of an element j is defined as $d_j = |I_j|$. Let $d = \max\{d_j, j = 1, \ldots, n\}$.

Due to its wide applications, the SUKP has recently received increased attention. An exact algorithm based on dynamic programming (Goldschmidt et al., 1994) has been presented to solve the SUKP. Because of the NP-hard performance of SUKP, this method is limited in application to very small problem instances.

In 2014, Arulselvan (2014) presented a greedy strategy based approximation algorithm (A-SUKP) for solving the SUKP. This greedy approach took into account all probable subsets with cardinality equal to 2 or lower, whose weight sum is within the capacity C. A-SUKP caused a gradual increase in every subitem in the subsets as the ratio $\frac{p_i}{W_i'}$ (where $W_i' = \sum_{j\in U_i} \frac{w_j}{d_j}$) decreased, if its inclusion did not violate the capacity C. The best of these augmented sets was returned as the approximation solution of the SUKP. The A-SUKP has approximation ratio $\frac{1}{1-e^{-\frac{1}{d}}}$. The approximation ratio reduces with increasing d. When d becomes large, A-SUKP has an undesirable approximation solution with low efficiency (He et al., 2018).

In 2018, He et al. developed a new BABC (He et al., 2018) to solve the SUKP. The BABC developed a novel food source updating strategy to generate new solutions. The food source updating strategy used a real vector to represent a solution, and defined surjection mapping to transform a continuous vector into a binary vector. In addition, BABC used a greedy repairing method to deal with infeasible solutions. The time complexity of the BABC is $O(M^4)$, where $M = max\{m, n\}$. The BABC was tested on 30 instances. Experimental results showed that BABC performs better than GA, ABC_{bin}, and binDE. However, from their experimental results, one can observe the averaged standard deviation (about 166.98) was relatively large, which implies that there is still scope for great improvement in the quality of the solutions BABC produces.

Due to their robustness and parallelism, population-based evolutionary algorithms have become powerful tools for solving various optimization problems. The SUKP is a generalization of the standard 0–1 knapsack problem. Many population-based evolutionary algorithms can be found for solving variants of knapsack problems (Changdar, Mahapatra, & Pal, 2015; Chen & Hao, 2016; Chen, Hao, & Glover, 2016; Haddar, Khemakhem, Hanafi, & Wilbaut, 2015; 2016; He, Wang, He, Zhao, & Li, 2016a; He, Zhang, Li, Wu, & Gao, 2016b; Meng & Pan, 2017). In contrast, the literature based on population-based evolutionary algorithms to the SUKP is quite poor.

Particle swarm optimization (PSO) is one of the most popular stochastic algorithms (Wang, Sun, Li, Rahnamayan, & Pan, 2013). It was first proposed by Eberhart and Kennedy (Kennedy & Eberhart, 1995), with an expectation that it would solve continuous problems. In order to solve discrete problems, Kennedy and Eberhart (1997) developed a discrete version of the PSO. Subsequently, different variants of discrete PSO have been successfully applied in various domains of combinatorial optimization (Jiang et al., 2017), such as feature selection (Chuang, Yang, & Li, 2011), the obnoxious p-median problem (Lin & Guan, 2018a), influence maximization (Gong, Yan, Shen, Ma, & Cai, 2016), the constrained shortest path problem (Marinakis, Migdalas, & Sifaleras, 2017), gene selection and cancer classification (Jain, Jain, & Jain, 2018), data allocation problem (Mahi, Baykan, & Kodaz, 2018), and cost sensitive attribute reduction (Dai, Han, Hu, & Liu, 2016). We have been motivated to devise a BPSO to solve the SUKP because discrete PSO has been used successfully in solving the difficult optimization problems.

This paper proposes a hybrid BPSO with tabu search (HBPSO/TS) to solve the SUKP and to obtain good quality solutions. The main contributions of this work are presented as follows:

1. The SUKP is a constrained binary programming problem. When applying the BPSOs to the SUKP, the fist issue to address is how to maintain the feasibility of candidate solutions during the search, i.e.,how to handle the constraints. The penalty function technique is the most simple and popular method to avoid the violation of the problem constraints. Because the value of penalty parameter is problem-dependent, it is hard to select a suitable value. To address this issue, we employ an adaptive penalty function to deal with the constraints. It is relatively easy to find a proper value of the penalty parameter in our proposed adaptive penalty function. In addition, this method makes the search focus on the boundary of the feasible solution space, and improves the efficiency.
2. Traditional discrete PSOs use different sigmoid functions (Kennedy & Eberhart, 1997) to generate new positions. These new positions produced by these position updating methods have good diversity. However, these position updating methods can not use the previously found solutions to guide the search. We redefine the method of updating positions based on the SUKP characteristics. With regard to combinatorial optimization problems, it is generally considered that high-quality solutions usually have a high degree of similarity. The new position updating method uses previously found results to generate new positions. In addition, we use a tabu based mutation procedure to diversify the search.
3. It is commonly acknowledged that local search significantly improves the performance of population-based evolutionary algorithms. The HBPSO/TS employs a tabu search procedure to improve the solution quality. In the tabu search that has been put forward, our self-adaption penalty function is used as the fitness function, so as to focus the search on the border between the feasible area and infeasible area. From the perspective of the quality of the solutions, the tabu search improves the algorithm performance markedly. Furthermore, a technology for updating gains has been put forward to lower the calculation expense.
4. From a computational perspective, our proposed algorithm outperforms existing approaches for solving the SUKP in terms of solution quality. Moreover, the suggested algorithm finds the new best solutions for 28 out of 30 instances. The average standard deviation of the proposed algorithm is 71.44, which is much smaller than that of the BABC (about 166.98). The results show that the HBPSO/TS is robust.

The rest of this article is structured as follows. Section 2 introduces an adaptive penalty function to deal with constraints. Section 3 describes the HBPSO/TS to solve the SUKP. Section 4 reports the computational results, and presents comparisons between HBPSO/TS and existing algorithms. Finally, Section 5 concludes with a summary of major results and suggestions for future researches.

2. The adaptive penalty function

The SUKP is a constrained binary programming problem. Our HBPSO/TS employs a tabu search to intensify the search. During the tabu search, it is important to maintain the feasibility of candidate solutions. The exact penalty function is one of the most popular penalty functions used to maintain feasibility during a search. Let $\eta(x)$ be the sum of constraints violation:

$$\eta(x) = max\left\{ \sum_{j\in\Lambda(x)} w_j - C, 0 \right\}. \tag{4}$$

Then, the exact penalty function can be defined as follows:

$$\omega(x,\theta) = f(x) - \theta \times \eta(x), \tag{5}$$

where $\theta > 0$ is a penalty parameter. It is commonly acknowledged that the parameter θ is problem-dependent (Ali & Zhu, 2013; Lin, Zhu, & Ali, 2016), and the value of θ is hard to determine.

In 2013, Ali and Zhu (2013) proposed an adaptive penalty function to deal with constraints for continuous optimization problems and showed that the value of the parameter in the adaptive penalty function is relatively easy to select. In this work, we employ the adaptive penalty function to deal with the constraints:

$$g(x,R) = \begin{cases} f(x) & if \quad \eta(x) = 0; \\ L - R \times \eta(x) & if \quad \eta(x) > 0 \quad and \quad f(x) \geq L; \\ f(x) - R \times \eta(x) & if \quad \eta(x) > 0 \quad and \quad f(x) < L, \end{cases} \tag{6}$$

in which, $R > 0$ is a penalty parameter, and L is inferior limit of the SKUP overall highest value. It is necessary to use the present best function value among the feasible solutions to update the L value.

Consider the following unconstrained binary optimization problem:

$$(UP)\begin{cases} \max & g(x,R) \\ s.t. & x \in \{0,1\}^m. \end{cases}$$

We make the following observation.

Observation 1. Suppose that L is a lower bound on the global maximum value of SUKP, if R is large enough, then SUKP and problem (UP) have the same global maximizers and global maximal values.

The HBPSO/TS uses $g(x, R)$ as a fitness function during the tabu search. Benefiting from the application of the adaptive penalty function $g(x, R)$, HBPSO/TS can maintain the feasibility of candidate solutions. Moreover, because of this, the particles will be encouraged to develop diversified feasible areas close to the boundary of the feasible zone, and to look for new and better solutions.

3. The proposed HBPSO/TS algorithm

This section describes the HBPSO/TS algorithm for the SUKP. First of all, the overall framework of HBPSO/TS is described. We then illustrate the main parts of HBPSO/TS.

3.1. General framework of the proposed HBPSO/TS

Kennedy and Eberhart (1995) first proposed PSO to solve continuous optimization problems. To solve discrete problems, Kennedy and Eberhart (1997) used a sigmoid function for the transformation of solutions into discrete space, and this gave rise to a discrete PSO. Subsequently, many variants of BPSOs have been developed for solving different discrete optimization problems. Recently, García and Pérez (2008) designed a novel discrete PSO method, the jumping PSO (JPSO), for solving combinatorial optimization problems. Because movement in the scattered space is not continuous, the concept of speed is meaningless; therefore, JPSO operates and maintains the appeals of the best locations free of the velocity component. Each particle has three attractors: its personal best position, the best position of its social neighborhood, and the global best position. During the search, each particle moves close to one of the attractors. Because of the high simplicity and great convenience in its operation, JPSO has been successfully used to set covering problem (Balaji & Revathi, 2016), and to set cover problem (Lin & Guan, 2018b).

Algorithm 1 General structure of HBPSO/TS.

Input: An instance of SUKP.
Output: The best solution *g_best* found.

1: **for** t from 1 to p **do**
2: Randomly generate an initial solution $x^t \in \{0, 1\}^m$.
3: $x^t \leftarrow$ *tabu search procedure* (x^t).
4: Initialize $p\_best^t = x^t$.
5: **end for**
6: Let $P = \{x^1, \cdots, x^p\}$, and $g\_best = argmax\{g(x^t, R), t = 1, \cdots, p\}$.
7: **while** (the maximum number of generations $G_{\max}$ has not be reached) **do**
8: Randomly select a particle x^t from P.
9: Generate a random number $\delta \in [0, 1)$.
10: **if** $0 \leq \delta < \frac{1}{3}$ **then**
11: $y \leftarrow p\_best^t$.
12: **end if**
13: **if** $\frac{1}{3} \leq \delta < \frac{2}{3}$ **then**
14: $y \leftarrow p\_best^l$, where l is randomly selected from$\{1, \cdots, p\}$, and $l \neq t$.
15: **end if**
16: **if** $\frac{2}{3} \leq \delta < 1$ **then**
17: $y \leftarrow g\_best$
18: **end if**
19: $x^t \leftarrow$ *position updating procedure* (x^t, y).
20: $x^t \leftarrow$ *tabu based mutation procedure* (x^t).
21: $x^t \leftarrow$ *tabu search procedure* (x^t).
22: **if** $g(x^t, R) > g(p\_best^t, R)$ and $x^t \neq p\_best^b, b = 1, \cdots, p$ **then**
23: Let $p\_best^t = x^t$.
24: **end if**
25: **if** $f(x^t) > f(g\_best)$ and $x^t \in \Omega$ **then**
26: Let $g\_best = x^t$, $L = f(x^t)$.
27: **end if**
28: **end while**
29: **return** *g_best* and $f(g\_best)$.

On the basis of a JPSO framework, we employ the detailed materials of the SUKP, redefine the position updating rule, and propose a hybrid BPSO with tabu search (HBPSO/TS) to solve SUKP.

As a natural evolution algorithm, HBPSO/TS is established on the basis of the population. Algorithm 1 provides the overall framework of HBPSO/TS to SUKP. HBPSO/TS is started from a preliminary swarm $P = \{x^1, \ldots, x^p\}$. Each preliminary solution x^t in the set P is produced in a random manner (line 2), and is then improved through a process of tabu search (line 3). Let p_best_t and *g_best* be the personal best position of particle t and the global best position of the swarm, respectively. p_best_t is initialized as $p\_best_t = x^t$ (line 4), and $g\_best = argmax\{g(x^t, R), x^t \in P\}$ (line 6). Subsequently, an iteration process is repeated until a fixed generations $G_{\max}$ is reached. At each generation, our HBPSO/TS first chooses a particle t at random (line 8). Next, x^t randomly selects one of the following attractors: the personal best position, other best positions, and the global best position. Specifically, we choose an attractor that is denoted by y in a random manner from

$\{p\_best_t, p\_best_l, g\_best\}$, where p_best_t, and p_best_l are the personal best positions of the tth and lth ($t \neq l$) particles, respectively; and g_best is the global best position. Then, x^t moves towards the chosen attractor y by the position updating procedure in line 19. To generate diversified solutions, a tabu based mutation procedure is employed (line 20). Finally, the newly generated solution x^t is optimized by the tabu search procedure in line 21. In order to maintain the difference, x^t is used to update p_best_t if $g(x^t, R) > g(p\_best_t, R)$, and x^t is not identical to $p\_best_j, j \in \{1, \ldots, p\}$ (lines 22–24). If x^t is feasible and better than g_best, we let $g\_best = x^t$ (lines 25–27).

Below, we describe the methods of using tabu search to strengthen the search, the working principles of the mutation program based on tabu, and the approach to update particle positions more clearly.

3.2. Tabu search procedure

Because local search can significantly improve the performance of a BPSO, we employ a tabu search procedure to refine the solutions obtained by the tabu based mutation procedure. The proposed tabu search procedure uses the traditional one-flip neighborhood $N(x)$. More formally, $N(x) = \{y \in \{0, 1\}^m : \sum_{i=1}^{m} |x_i - y_i| \leq 1\}$.

Let y be the best solution found so far. Algorithm 2 presents the tabu search procedure. It begins with an initial solution x. Some passes are included in the tabu search process, with every pass containing $I_{\max}$ iterations. At the beginning of a pass, all variables can be flipped freely (line 4). In each iteration, we flip a variable according to the definition of flip gain. The *gain*(i, x) of a variable flip gain refers to the augmentation of the fitness function brought by the variable x_i flipping. In paticular, we make the following definition:

$$gain(i, x) = g(x', R) - g(x, R), \tag{7}$$

where $x' = (x_1, \ldots, x_{i-1}, 1 - x_i, x_{i+1}, \ldots, x_m)$. Line 2 calculates the flip gains using (7). The tabu search procedure then iteratively chooses a free x_i that has the maximum flip gain in the $N(x)$ for being flipped (lines 6–7). Once a flip is performed, we update the flip gains of unlocked variables. We need $O(mn)$ to calculate $g(x, R)$. Hence, it takes $O(mn \times m)$ to update all unlock flip gains by (7). This process is time-consuming. Therefore, a gain updating technology is used to update flip gains more quickly (line 8). This is illustrated in the following paragraphs.

Algorithm 2 Tabu search procedure.

Input: An initial solution x.
Output: The optimized solution y.

1: Initialize $y = x$.
2: Calculate the flip gains $gain(i, x), i = 1, \cdots, m$, according to (7). Let $flag = 1$.
3: **while** $flag = 1$ **do**
4: Initialize $\kappa(i) = 0, i = 1, \cdots, m$, and let $flag = 0$.
5: **while** (the maximum number of iterations $I_{\max}$ has not be reached) **do**
6: Let $t = argmax\{gain(i, x), \kappa(i) = 0\}$.
7: Let $x_t = 1 - x_t$.
8: Use the gain update technique to update the flip gains.
9: **for** i from 1 to p **do**
10: **if** $\kappa(i) > 0$ **then**
11: $\kappa(i) = \kappa(i) - 1$.
12: **end if**
13: Let $\kappa(t) = \mu + rand(10)$.
14: **if** $g(x, R) > g(y, R)$ **then**
15: Let $y = x$, and $flag = 1$.
16: **end if**
17: **end for**
18: **end while**
19: **end while**
20: **if** $g(y, R) > g(g\_best, R)$ and $x \in \Omega$ **then**
21: $g\_best = y, L = f(y)$.
22: **end if**
23: **return** y.

After a variable x_i is flipped, it is not allowed to flip the variable x_i in the following $\kappa(i)$ iterations (line 13). For this study, we set:

$$\kappa(i) = \mu + rand(10),$$

where μ is a given constant and rand(10) takes a random value from 1 to 10. In case the newly obtained solution is more optimized than y, we replace y by x (lines 14–16). The above process is repeated with each pass of the tabu search program until the number of iterations $I_{\max}$, which is defined in advance, is reached. In addition, the best solution in a pass is used as the beginning solution in the following pass. When a pass can not find a more optimized solution, the tabu search process comes to an end. Finally, if $g(y, R) > g(g\_best, R)$, and y is a feasible solution of SUKP, we update the global best position g_best and the lower bound L (lines 19–22).

3.3. Gain updating method

When the tabu search flips a variable, the flip gains of other variables may be changed. To reduce computational cost, the flip gains are updated by the gain updating technology rather than by definition of the flip gain. In tabu search process, we keep a vector ($s_1, \cdots, s_n$), where s_j reserves the time of j that is chosen by x, so as to rapidly update the influenced flip gains. More formally, we define:

$$s_j = |T_j|, \tag{8}$$

where $T_j = \{i \in S : j \in U_i, x_i = 1\}$ is the set of selected items which contain element j.

Suppose that the current candidate solution is x. Let *current_W* and *current_P* be the current weight sum and the profit sum of x, respectively. Specifically, $current\_W = \sum_{j \in \Lambda(x)} w_j$, and $current\_P = f(x)$. The basic idea of the gain updating technique is to compute the fitness function value $g(x, R)$ by *current_W* and *current_P* quickly, and then the affected flip gains are updated by (7).

When the tabu search has just starts, every s_j, which is the time of element j chosen by x, is calculated by (8). In addition, (7) is used to calculate the flip gains. Suppose that the variable x_t is selected to flip. Once variable x_t has been flipped, we need to update the flip gains.

Algorithm 3 provides an illustration of the gain updating technology. We first analyze the updating of the current weight sum and the current profit sum after x_t is flipped. Two cases are considered:

Case 1: If $x_t = 1$ (line 1), item t is added into the knapsack, and then the current profit sum increases by p_t (line 2); Because all elements in U_t are added into the knapsack, for any element $j \in U_t$, the time of element j selected by x increases by one, i.e., $s_j = s_j + 1$ (lines 3–4). $s_j = 1$ indicates that the element j is newly added into the knapsack. So, if $s_j = 1$, we increase the current weight sum by w_j (line 6).

Case 2: If $x_t = 0$ (line 9), item t is removed from the knapsack, and then the current profit sum decreases by p_t (line 10). Because all elements in U_t are removed from the knapsack, for any element $j \in U_t$, the time of element j selected by x decreases by one, i.e., $s_j = s_j - 1$ (lines 11–12). For each element $j \in U_t$, $s_j = 0$ indicates that element j is removed form the knapsack after this flipping. So, if $s_j = 0$, the current weight sum decreases by w_j (line 14).

Algorithm 3 Gain updating technique.

Input: An initial solution x, $g(x, R)$, the last flipped variable x_t, $current\_P$, $current\_W$, $gain(i, x), i = 1, \cdots, m$ and $s_j, j = 1, \cdots, m$.
Output: The updated flip gains.
1: **if** $x_t = 1$ **then**
2: $current\_P = current\_P + p_t$.
3: **for** each element $j \in U_t$ **do**
4: $s_j = s_j + 1$.
5: **if** $s_j = 1$ **then**
6: $current\_W = current\_W + w_j$.
7: **end if**
8: **end for**
9: **else**
10: $current\_P = current\_P - p_t$.
11: **for** each element $j \in U_t$ **do**
12: $s_j = s_j - 1$.
13: **if** $s_j = 0$ **then**
14: $current\_W = current\_W - w_j$.
15: **end if**
16: **end for**
17: **end if**
18: **for** each free item k **do**
19: Let $temporary\_P(x) = current\_P$, and $temporary\_W(x) = current\_W$.
20: **if** $x_k = 1$ **then**
21: $temporary\_P(x) = temporary\_P(x) - p_k$.
22: **for** each element $j \in U_k$ **do**
23: **if** $s_j = 1$ **then**
24: $temporary\_W(x) = temporary\_W(x) - w_j$.
25: **end if**
26: **end for**
27: **else**
28: $temporary\_P(x) = temporary\_P(x) + p_k$.
29: **for** each element $j \in U_k$ **do**
30: **if** $s_j = 0$ **then**
31: $temporary\_W(x) = temporary\_W(x) + w_j$.
32: **end if**
33: **end for**
34: **end if**
35: Let $f(x) = temporary\_P(x), \eta(x) = max\{temporary\_W(x) - C, 0\}$, then $g(x', R)$ can be computed by (6) directly, where $x' = (x_1, \cdots, 1 - x_k, \cdots, x_n)$.
36: Let $gain(k, x) = g(x', R) - g(x, R)$.
37: **end for**
38: **return** The updated flip gains.

Let $temporary\_P(x)$, and $temporary\_W(x)$ be the profit sum, and the weight sum, respectively, after we flip x_k. To compute the flip gain of each free item, we need to calculate $temporary\_P(x)$, and $temporary\_W(x)$. For each free item k, $temporary\_P(x)$, and $temporary\_W(x)$ are initialized as the current profit sum and the current weight sum (line 19), respectively. Similar to the above updating process, two cases are considered.

Case 1: If $x_k = 1$ (line 20), flipping x_k means that item k will be removed from the knapsack, then, flipping x_k will decrease the profit sum (line 21). Because all elements in U_k will be removed from the knapsack, for any element $j \in U_k$, the time of element j selected by x decreases by one. If $s_j = 1$ (line 23), the weight sum is updated by $temporary\_W(x) = temporary\_W(x) - w_j$ (line 24).

Case 2: If $x_k = 0$ (line 27), flipping x_k means that item k will be added into the knapsack, then, flipping x_k will increase the profit sum (line 28). Because all elements in U_k will be added into the knapsack, for any element $j \in U_k$, the time of element j selected by x increases by one. If $s_j = 0$ (line 30), the current weight sum is update by $temporary\_W(x) = temporary\_W(x) + w_j$ (line 31).

In lines 35–36, the gain updating technique uses $temporary\_P(x)$, and $temporary\_W(x)$ to calculate $gain(k, x)$ according to (6), and (7).

We need $O(d)$ to update $current\_P$ and $current\_W$ from line 1 to 17. For each unlock variable, it takes $O(d)$ to calculate the new flip gain from line 19 to 34. Hence, the total time complexity of using the gain updating technique is bound by $O(md)$, which is substantially lower than the time complexity of using the definition of the flip gain.

3.4. Position updating procedure and tabu based mutation procedure

Many studies (Lin et al., 2016; Lin & Zhu, 2014; Wu & Hao, 2013) have concluded that the high-quality solutions for problems related to combination optimization have very small distances. In order to guide the search to focus on a region near high quality solutions, the position updating procedure tries to move the selected particle close to one of the high quality solutions.

Let $x^t = (x_1^t, \ldots, x_m^t)$ and $y = (y_1, \ldots, y_m)$ be the selected particle and the selected attractor, respectively. Let z be the newly generated solution. Algorithm 4 shows the pseudo-code of the position updating procedure. If item i is selected by both x^t and y, or item i is not selected by both x^t and y, i.e., $x_i^t = y_i$, then we let $z_i = y_i$. Otherwise, a number $\delta \in (0, 1)$ is randomly generated, if $\delta < 0.5$, and we let $z_i = y_i$; otherwise, we let $z_i = x_i^t$.

At the beginning of the search, the particles have relatively large distances with the personal best positions, and the position updating procedure can lead the search to enter into new hopeful regions. As the search progress, the degrees of similarity of particles and the personal best positions are large, and new solutions generated by the position updating procedure may be very similar to the personal best positions. As a result, the raised HBPSO/TS will converge prematurely. With the aim of eliminating this disadvantage, we design a mutation procedure based on tabu that generates diverse solutions by flipping several variables.

HBPSO/TS keeps a short-run memory to avoid recently flipped variables being flipped again in the following generations. More formally, we maintain a vector $d = (d_1, \ldots, d_m)$. At the beginning of HBPSO/TS, all variables are free to flip, i.e., we initialize $d_i = 0, i = 1, \ldots, m$. z is supposed to be a new solution produced in the position updating process when the HBPSO/TS is at the kth generation. $d_i \leq k$ indicates that the variable x_i can be flipped; otherwise, the variable x_i is forbidden to be flipped.

Algorithm 4 Position updating procedure.

Input: A selected solution x, and a selected attractor y.
Output: A newly generated solution z.
for i from 1 to m **do**
if $x_i^t = y_i$ **then**
$z_i = y_i$
else
Randomly generate a $\delta \in (0, 1)$.
if $\delta < 0.5$ **then**
$z_i = y_i$
else
$z_i = x_i^t$
end if
end if
end for
return The new solution z.

Table 1
The numbering of all SUKP instances.

ID	The first set ($m > n$)	ID	The second set ($m = n$)	ID	The third set ($m < n$)
Fi1	sukp 100_85_0.1_0.75	Si1	sukp 100_100_0.1_0.75	Ti1	sukp 85_100_0.1_0.75
Fi2	sukp 100_85_0.15_0.85	Si2	sukp 100_100_0.15_0.85	Ti2	sukp 85_100_0.15_0.85
Fi3	sukp 200_185_0.1_0.75	Si3	sukp 200_200_0.1_0.75	Ti3	sukp 185_200_0.1_0.75
Fi4	sukp 200_185_0.15_0.85	Si4	sukp 200_200_0.15_0.85	Ti4	sukp 185_200_0.15_0.85
Fi5	sukp 300_285_0.1_0.75	Si5	sukp 300_300_0.1_0.75	Ti5	sukp 285_300_0.1_0.75
Fi6	sukp 300_285_0.15_0.85	Si6	sukp 300_300_0.15_0.85	Ti6	sukp 285_300_0.15_0.85
Fi7	sukp 400_385_0.1_0.75	Si7	sukp 400_400_0.1_0.75	Ti7	sukp 385_400_0.1_0.75
Fi8	sukp 400_385_0.15_0.85	Si8	sukp 400_400_0.15_0.85	Ti8	sukp 385_400_0.15_0.85
Fi9	sukp 500_485_0.1_0.75	Si9	sukp 500_500_0.1_0.75	Ti9	sukp 485_500_0.1_0.75
Fi10	sukp 500_485_0.15_0.85	Si10	sukp 500_500_0.15_0.85	Ti10	sukp 485_500_0.15_0.85

Algorithm 5 Tabu based mutation procedure.

Input: A solution z, the kth generation.
Output: The newlygenerated solution z.

for i from 1 to m **do**
  if $d_i \le k$ **then**
    Randomly generate a number $\delta \in (0, 1)$.
    if $\delta \le p_{mu}$ **then**
      $z_i = 1 - z_i$, $d_i = k + \lambda$.
    end if
  end if
end for
return The new solution z.

Algorithm 5 describes the tabu based mutation procedure. At the kth generation of HBPSO/TS, the tabu based mutation procedure starts with z, and each free variable x_i (i.e., $d_i \le k$) is flipped with a probability p_{mu}. If a free variable is flipped, it is forbidden to flip again in the next λ generations. Specially, we let $d_i = k + \lambda$.

The mutation procedure established on the basis of tabu succeeds in leading to enter a new hopeful area.

3.5. Time complexity of HBPSO/TS

The HBPSO/TS has three main procedures: the tabu search procedure, the position updating procedure, and the tabu based mutation procedure.

The tabu search procedure uses $O(m^2 n)$ to calculate the initial flip gains. The tabu search procedure then iteratively flips a variable. In each pass, $O(m)$ is used by the tabu search process to discern the variable that has the highest flip gain, and $O(md)$ is used to update the flip gains. The tabu search process includes $I_{\max}$ iterations in one pass. Hence, the overall time complexity of the tabu search process in one pass is limited by $O(I_{\max} mn + m^2 n)$. The position updating procedure takes $O(m)$ to produce a new position, and the time complexity of the tabu based mutation procedure is $O(m)$.

Therefore, the time complexity of one generation of HBPSO/TS is $O(I_{\max} mn + m^2 n + 2m)$, and the total time complexity of HBPSO/TS is $O((I_{\max} mn + m^2 n + 2m) \times G_{\max})$.

4. Results[1]

He et al. (2018) proposed three sets of 30 instances to test their proposed BABC. We also use these instances to test our HBPSO/TS.

Each instance is associated with a binary matrix $M = (r_{ij})$, which defines subset family $\{U_1, \cdots, U_m\}$. For each $r_{ij} (i = 1, \ldots, m; j = 1, \ldots, n)$ in M, $r_{ij} = 1$ if and only if $j \in U_i$. Each instance is named as sukp $m\_n\_\alpha\_\beta$, where $\alpha = (\sum_{i=1}^{m} \sum_{j=1}^{n} r_{ij})/(mn)$, and $\beta = C / \sum_{j=1}^{n} w_j$ are the density of the matrix M, and the ratio of C to the sum of all elements, respectively. The tested instances can be classed as three sets. Based on the above naming rules, all the tested instances are indexed and reported in Table 1. All these instances can be found in http://sncet.com/ThreekindsofSUKPinstances(EAs).rar.

[1] This section contains an experimental assessment of the suggested HBPSO/TS. The implementation of our HBPSO/TS was performed in C and operated on a computer with 3.4GHz processor (i7 6700) and 8GB of RAM.

Table 2
Results on the Friedman test ($\alpha = 0.05$).

	R	p	$G_{\max}$	μ	$I_{\max}$	p_{mu}	λ
p-value	0.129	0.129	0.080	0.354	0.760	0.171	0.125

Table 3
Settings of important parameters of HBPSO/TS.

Parameters	Section	Value
R	3	2
p	4.1	20
$G_{\max}$	4.1	0.6m
μ	4.2	8
$I_{\max}$	4.2	0.6m
p_{mu}	4.4	0.2
λ	4.4	15

4.1. Preliminary experiments for parameters

We conducted some preliminary experiments to obtain suggestions for selecting the parameters. A representative subset with three instances was used for determining our parameter values: sukp 500_485_0.15_0.85 (Fi10), sukp 500_500_0.15_0.85 (Si10), and sukp 485_500_0.15_0.85 (Ti10).

Firstly, a large range of values for each parameter was tested to find an interval of reasonable size. Then, we focused on these relatively suitable values. More formally, we tested R in the scope [2,10], p in the scope [5,30], $G_{\max}$ in the scope [0.1n, 2n], μ in the range [3,15], $I_{\max}$ in the scope [0.4n, 2n], p_{mu} in the range [0.08,0.3], and λ in the range [8,30].

The Friedman test was used to check whether the HBPSO/TS performance changes significantly with regards to the mean objective function values when we changed the value of one parameter as stated. The p-values obtained from the Friedman test are shown in Table 2, from which, we conclude that these parameters have little influence on the HBPSO/TS performance. We selected the values of these parameters using their rankings from the Friedman's test. Table 3 summarizes the selected values of the parameters.

Table 4
The computing results of the first set of SUKP instances.

Instance	Results	A-SUKP	GA	BABC	ABC_{bin}	binDE	HBPSO/TS	improve
sukp 100_85_0.1_0.75	*Best*	12459	13044	13251	13044	13044	**13283**	0.24%
	Mean	12459	12956.4	13028.5	12818.5	12991	**13277.03**	1.90%
sukp 100_85_0.15_0.85	*Best*	11119	12066	12238	12238	12274	**12348**	0.60%
	Mean	11119	11546	12155	12049.3	12123.9	**12262.25**	0.88%
sukp 200_185_0.10_0.75	*Best*	11292	13064	13241	12946	13241	**13521**	2.11%
	Mean	11292	12492.5	13064.4	11861.5	12940.7	**13521.0**	3.49%
sukp 200_185_0.15_0.85	*Best*	12262	13671	13829	13671	13671	**14215**	2.79%
	Mean	12262	12802.9	13359.2	12537	13110	**13952.8**	4.44%
sukp 300_285_0.10_0.75	*Best*	8941	10553	10428	9751	10420	**11563**	9.57%
	Mean	8941	9980.87	9994.76	9339.3	9899.24	**11401.93**	14.23%
sukp 300_285_0.15_0.85	*Best*	9432	11016	12012	10913	11661	**12607**	4.95%
	Mean	9432	10349.8	10902.9	9957.85	10499.4	**12457.73**	14.26%
sukp 400_385_0.10_0.75	*Best*	9076	10083	10766	9674	10576	**11484**	6.66%
	Mean	9076	9641.85	10065.2	9187.76	9681.46	**11439.75**	13.65%
sukp 400_385_0.15_0.85	*Best*	8514	9831	9649	8978	9649	**11209**	14.01%
	Mean	8514	9326.77	9135.98	8539.95	9020.87	**11031.7**	18.27%
sukp 500_485_0.10_0.75	*Best*	9864	11031	10784	10340	10586	**11716**	6.20%
	Mean	9864	10567.9	10452.2	9910.32	10363.8	**11484.65**	8.67%
sukp 500_485_0.15_0.85	*Best*	8299	9472	9090	8759	9191	**10194**	7.62%
	Mean	8299	8692.67	8857.89	8365.04	8783.99	**9896.875**	11.72%

Table 5
The computing results of the second set of SUKP instances.

Instance	Results	A-SUKP	GA	BABC	ABC_{bin}	binDE	HBPSO/TS	improve
sukp 100_100_0.10_0.75	*Best*	13634	**14044**	13860	13860	13814	13990	-0.38%
	Mean	13634	13806	13734.9	13547.2	13675.9	**13952.8**	1.06%
ukp 100_100_0.15_0.85	*Best*	11325	13145	**13508**	13498	13407	13498	-0.07%
	Mean	11325	12234.8	**13352.4**	13103.1	13212.8	13293.45	-0.44%
sukp 200_200_0.10_0.75	*Best*	10328	11656	11846	11191	11535	**12522**	5.70%
	Mean	10328	10888.7	11194.3	10424.1	10969.4	**12498.98**	11.65%
sukp 200_200_0.15_0.85	*Best*	9784	11792	11521	11287	11469	**12317**	4.45%
	Mean	9784	10827.5	10945	10345.9	10717.1	**12299.6**	13.59%
sukp 300_300_0.10_0.75	*Best*	10208	12055	12186	11494	12304	**12736**	4.51%
	Mean	10208	11755.1	11945.8	10922.3	11864.4	**12715.58**	6.44%
sukp 300_300_0.15_0.85	*Best*	9183	10666	10382	9633	10382	**11585**	8.61%
	Mean	9183	10099.2	9859.69	9186.87	9710.37	**11532.08**	14.18
sukp 400_400_0.10_0.75	*Best*	9751	10570	10626	10160	10462	**11433**	7.59%
	Mean	9751	10112.4	10101.1	9549.04	9975.8	**11399.65**	12.72%
sukp 400_400_0.15_0.85	*Best*	8497	9235	9541	9033	9388	**11325**	20.63%
	Mean	8497	8793.76	9032.95	8365.62	8768.42	**11321.0**	25.33%
sukp 500_500_0.10_0.75	*Best*	9615	10460	10755	10071	10546	**10973**	2.02%
	Mean	9615	10185.4	10328.5	9738.17	10227.7	**10832.53**	4.87%
sukp 500_500_0.15_0.85	*Best*	7883	9496	9318	9262	9312	**10086**	6.21%
	Mean	7883	8882.88	9180.74	8617.91	9096.13	**9797.25**	6.71%

4.2. Computational results and contrast to other algorithms

In this subsection, we report on the experiments on the three sets of 30 instances. The settings of the parameters in Table 3 are used in the HBPSO/TS. It is important to note that the computational results can be improved by special adjustments.

With an expectation to display the HBPSO/TS effectiveness, HBPSO/TS is compared with BABC (He et al., 2018), genetic algorithm (GA) (Schmitt, 2001), binary differential evolution (binDE) (Engelbrecht & Pampara, 2007), and continuous artificial bee colony algorithm (ABC_{bin}) (Kiran, 2015). All above algorithms were implemented in C++ on a computer with an (i5-3337u) 1.8GHz processor and 4GB of RAM.

We ran the HBPSO/TS with the parameter values in Table 3. The HBPSO/TS was operated 40 times for all 30 instances. Tables 4, 5, and 6 show the best result (*Best*), the average result (*Mean*), the standard deviation (*StD*) over the 40 repetitions, and the mean solution time (*Time*) in seconds. The data for BABC, GA, binDE, and ABC_{bin} are taken directly from (He et al., 2018). The column 'improve' lists the percentage deviations between the results obtained from HBPSO/TS and the previous best values accordingly. The best objective function values of the instances obtained from the algorithms are shown in bold in Tables 4, 5, and 6.

According to Tables 4, 5, and 6, we make the following observations.

(1) Our HBPSO/TS improved the previous best known results for 27 of the 30 instances, and matched the previous best known results for one instance. Moreover, the improvements with regards to objective function value raged from 0.24% to 20.63%.
(2) HBPSO/TS produced better mean objective function values in 29 of the 30 instances. The improvements in terms of objective function value ranged from 0.88% to 25.33%.
(3) HBPSO/TS performed better than A-SUKP, ABC_{bin}, and binDE for all tested instances with regards to the best objective function value and the mean objective function value. HBPSO/TS found better solutions for all instances, except for sukp 100_100_0.10_0.75.
(4) HBPSO/TS performed better than BABC for all instances, except sukp 100_100_0.15_0.85, in terms of the best objective function values and the mean objective function values.

To further test the statistical significance of our results, we adopted a non-parametric statistical test (Derrac, García, Molina, & Herrera, 2011; García, Molina, Lozano, & Herrera, 2009) to compare the HBPSO/TS with A-SUKP, GA, ABC_{bin}, and binDE.

Table 6
The computing results of the third set of SUKP instances.

Instance	Results	A-SUKP	GA	BABC	ABC_{bin}	binDE	HBPSO/TS	improve
sukp 85_100_0.10_0.75	*Best*	10231	11454	11664	11206	11352	**12045**	3.26%
	Mean	10231	11092.7	11182.7	10879.5	11075	**12045.0**	7.71%
sukp 85_100_0.15_0.85	*Best*	10483	12124	**12369**	12006	**12369**	**12369**	0.00%
	Mean	10483	11326.3	12081.6	11485.3	11875.9	**12335.75**	2.10%
sukp 185_200_0.10_0.75	*Best*	11508	12841	13047	12308	13024	**13696**	4.97%
	Mean	11508	12236.6	12522.8	11667.9	12277.5	**13690.8**	9.32%
sukp 185_200_0.15_0.85	*Best*	8621	10920	10602	10376	10547	**11298**	3.46%
	Mean	8621	10351.5	10150.6	9684.33	10085.4	**11298.0**	9.14%
sukp 285_300_0.10_0.75	*Best*	9961	10994	11158	10269	11152	**11802**	5.77%
	Mean	9961	10640.1	10775.9	9957.09	10661.3	**11787.2**	9.38%
sukp 285_300_0.15_0.85	*Best*	9618	11093	10528	10051	10528	**11538**	4.01%
	Mean	9618	10190.3	9897.92	9424.15	9832.32	**11536.78**	13.21%
sukp 385_400_0.10_0.75	*Best*	8672	9799	10085	9235	9883	**10465**	3.76%
	Mean	8672	9432.82	9537.5	8904.94	9314.57	**10340.28**	8.41%
sukp 385_400_0.15_0.85	*Best*	8064	9173	9456	8932	9352	**10506**	11.10%
	Mean	8064	8703.66	9090.03	8407.06	8846.99	**10339.45**	13.74%
sukp 485_500_0.10_0.75	*Best*	9559	10311	10823	10357	10728	**11115**	2.69%
	Mean	9559	9993.16	10483.4	9615.37	10159.4	**10872.38**	3.71%
sukp 485_500_0.15_0.85	*Best*	8157	9329	9333	8799	9218	**10104**	8.26%
	Mean	8157	8849.46	9085.57	8347.82	8919.64	**10008.13**	10.15%

Table 7
Results of the Friedman and Iman-Davenport tests ($\alpha = 0.05$).

	Friedman value	*p*-value	Iman-Davenport Value	*p*-value
Best	123.357	8.864E-11	134.270	1.381E-52
Mean	129.919	6.847E-11	187.623	1.110E-16

Table 8
The rankings obtained by Friedman's test.

Algorithm	*Best*	*Mean*
A-SUKP	5.966	5.783
GA	3.266	3.516
BABC	2.500	2.233
ABC_*bin*	4.750	5.000
binDE	3.400	3.433
HBPSO/TS	1.116	1.033

Table 9
p-values with respect to the best objective function value (HBPSO/TS is the control algorithm).

HBPSO/TS vs.	*z*	Unadjusted *p*	Holm *p*	Hochberg *p*
A-SUKP	10.040	1.012	5.06E-23	5.06E-23
GA	4.450	8.550	1.71E-5	1.71E-5
BABC	2.863	0.004	0.004	0.004
ABC_*bin*	7.521	5.406	2.16E-13	2.16E-13
binDE	4.726	2.279	6.83E-6	6.83E-6

Table 10
p-values with respect to the average objective function value (HBPSO/TS is the control algorithm).

HBPSO/TS vs.	*z*	Unadjusted *p*	Holm *p*	Hochberg *p*
A-SUKP	9.833	8.08E-23	4.04E-22	4.04E-22
GA	5.140	2.73E-7	8.19E-7	8.19E-7
BABC	2.484	0.012	0.012	0.012
ABC_*bin*	8.211	2.17E-16	8.71E-16	8.17E-16
binDE	4.968	6.74E-7	1.34E-6	1.34E-6

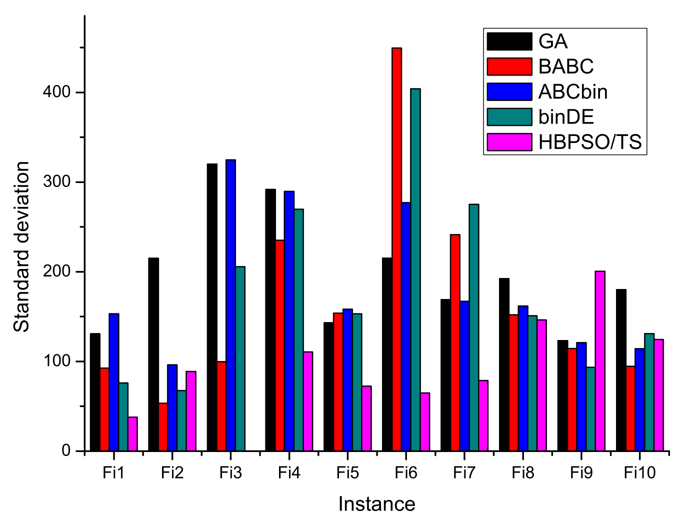

Fig. 1. The standard deviation of GA, BABC, ABC_{bin}, binDE, and HBPSO/TS for solving the first set of SUKP instances.

First, the Iman-Davenport and Friedman tests were used to analyze the differences between the results from the two algorithms. The results of the Friedman and Iman-Davenport tests on the best results (*Best*) and the average results (*Mean*) are provided in Table 7. There were significant differences ($p < .05$) in the results in terms of best objective function values and mean objective values. The rankings from Friedman test are displayed in Table 8, and show that HBPSO/TS which is ranked the lowest has the highest effectiveness of the six algorithms.

Stronger analytical procedures (Holm's and Hochberg's procedures (García et al., 2009)) were then utilized to compare the control algorithm HBPSO/TS with A-SUKP, GA, BABC, ABC_{bin}, and binDE. Holm's and Hochberg's procedures were established on the basis of the calculation of the adjusted *p*-values. Tables 9, and 10 summarize the *p*-values from these procedures for the best objective function value, and the average objective function value, respectively.

From Tables 9 and 10, it is observed that HBPSO/TS significantly outperforms A-SUKP, GA, BABC, ABC_{bin}, and binDE with regards to solution quality. However, the solution time of HBPSO/TS was longer than those of GA, BABC, ABC_{bin}, and binDE.

Next, we compare the stability of HBPSO/TS with that of other algorithms. Figs. 1, 2, and 3 show the standard deviation (StD) of GA, BABC, ABC_{bin}, binDE, and HBPSO/TS for solving three sets of SUKP instances, respectively. The standard deviation of HBPSO/TS was the smallest among these algorithms in 7 of the 10 instances in the first set, in 7 of the 10 instances in the second set, and in

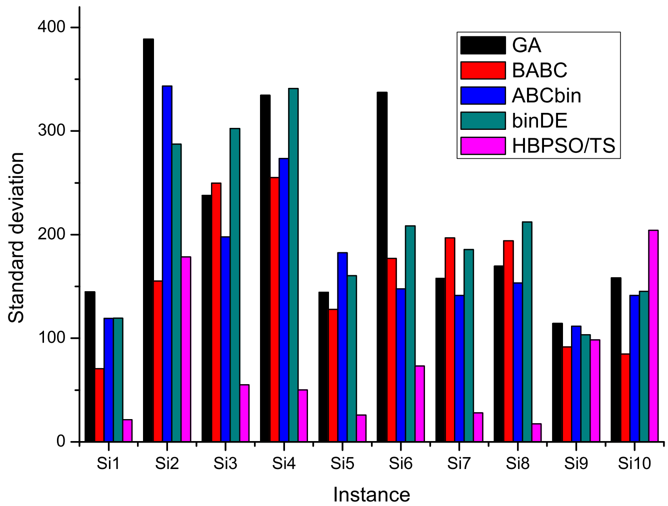

Fig. 2. The standard deviation of GA, BABC, ABC_{bin}, binDE, and HBPSO/TS for solving the second set of SUKP instances.

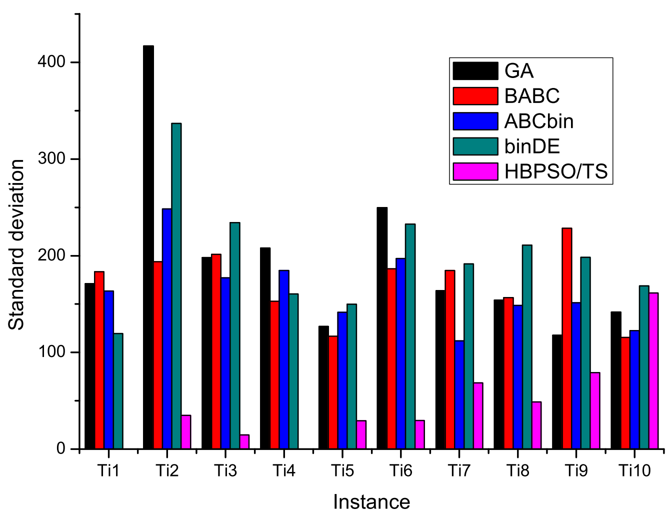

Fig. 3. The standard deviation of GA, BABC, ABC_{bin}, binDE, and HBPSO/TS for solving the third set of SUKP instances.

9 of the 10 instances in the third set (Figs. 1, 2, and 3). Moveover, the average standard deviations of GA, BABC, ABC_{bin}, binDE, and HBPSO/TS were 203.86, 166.98, 177.37, 196.54, and 71.44, respectively. The average standard deviation of HBPSO/TS was much smaller than those of the other heuristic algorithms, showing the high stability of the HBPSO/TS.

Recently, a check and repair operator-inspired particle swarm optimization with the neighborhood local search (3R-SACRO-PSO) (Chih, 2018) was proposed for solving the multidimensional knapsack problem. 3R-SACRO-PSO presented three pseudo-utility ratios to repair infeasible solutions, and used a velocity and position updating rule to generate new positions. In addition, a neighborhood local search was developed to improve the solution quality. Extensive experiment was done to validate the performance of 3R-SACRO-PSO.

Finally, we compare our proposed algorithm with a variant of 3R-SACRO-PSO (called VR-SACRO-PSO for short). Different from 3R-SACRO-PSO, the VR-SACRO-PSO uses only one pseudo-utility ratio. According to the structure of the SUKP, the pseudo-utility ratio δ_i used in VR-SACRO-PSO is defined as follows: $\delta_i = \frac{p_i}{\sum_{j\in U_i} w_j}$. We set the number of particles as 200, and other values of parameters of VR-SACRO-PSO are set as the same as those in (Chih, 2018).

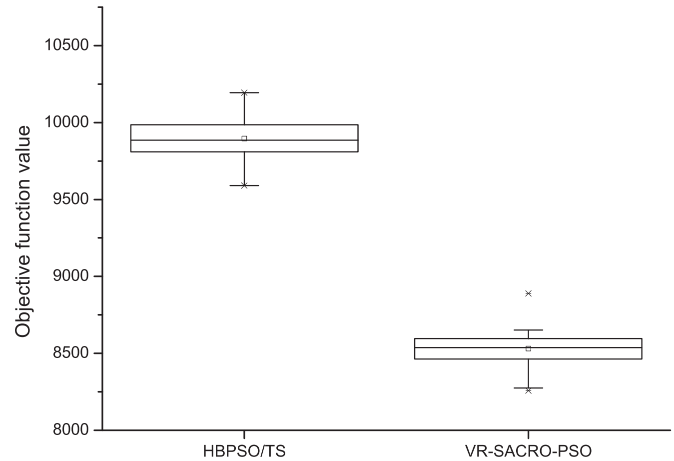

Fig. 4. Boxplot of the objective function values obtained by HBPSO/TS and VR-SACRO-PSO on Fi10.

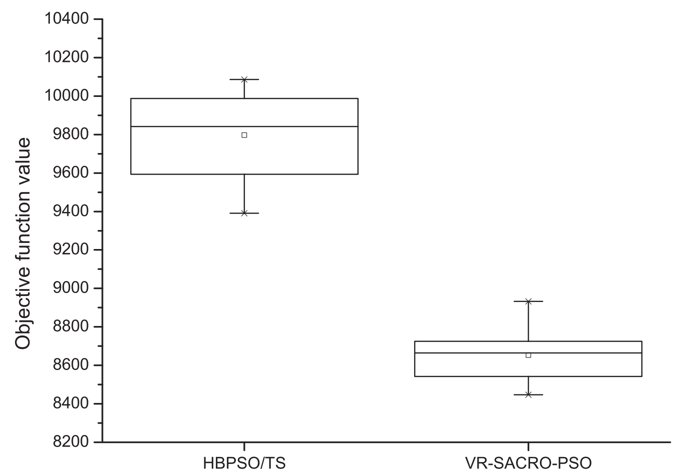

Fig. 5. Boxplot of the objective function values obtained by HBPSO/TS and VR-SACRO-PSO on Si10.

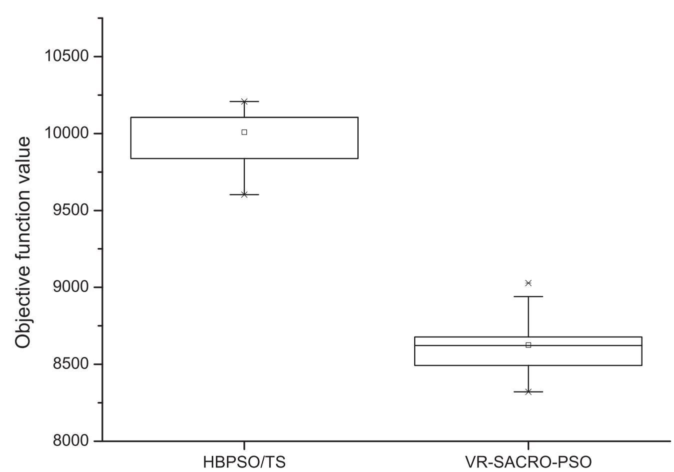

Fig. 6. Boxplot of the objective function values obtained by HBPSO/TS and VR-SACRO-PSO on Ti10.

We ran the VR-SACRO-PSO 40 times on three instances: sukp 500_485_0.15_0.85 (Fi10), sukp 500_500_0.15_0.85 (Si10), and sukp 485_500_0.15_0.85 (Ti10).

Figs. 4–6 plot the boxplots of the objective function value obtained through our proposed algorithm and VR-SACRO-PSO for

Table 11
Experimental results of HBPSO/TS, and HBPSO/TS1 with different R, and θ.

	Fi10	Si10	Ti10
HBPSO/TS ($R=2$)	9896.875	9797.250	10008.130
HBPSO/TS ($R=4$)	9626.875	9728.775	9685.275
HBPSO/TS ($R=10$)	9577.225	9559.750	9619.825
HBPSO/TS1 ($\theta=2$)	9878.500	9619.400	9918.850
HBPSO/TS1 ($\theta=4$)	9647.325	9437.293	9647.975
HBPSO/TS1 ($\theta=10$)	9302.488	9321.854	9583.125

Fi10, Si10, and Ti10, respectively. From Figs. 4–6, we can see that HBPSO/TS performs better than VR-SACRO-PSO.

4.3. Effectiveness of the key components of HBPSO/TS

In this section, we analyze and discuss the important aspects of HBPSO/TS, i.e., the adaptive penalty function, the tabu search procedure, and the tabu based mutation procedure.

First, we carried out a test to compare HBPSO/TS and one of its variants to ascertain whether the adaptive penalty function is effective. HBPSO/TS1 was used to represent the HBPSO/TS with the specific penalty function defined in (5). We ran HBPSO/TS, and HBPSO/TS1 40 times with the parameter values listed in Table 3 and with different R, and θ on sukp 500_485_0.15_0.85 (Fi10), sukp 500_500_0.15_0.85 (Si10), and sukp 485_500_0.15_0.85 (Ti10).

From Table 11, one can see that HBPSO/TS with $R=2$ performed better than all versions of HBPSO/TS1. When the penalty parameter has its value being increased from 2 to 10, the mean objective function values obtained by HBPSO/TS and HBPSO/TS1 decreased. In addition, with the increase in the value of R, and θ, the average objective function values obtained by HBPSO/TS changed more slowly than those of HBPSO/TS1, which indicates that it is relatively easy to choose a suitable value for R in the adaptive penalty function.

The tabu based mutation procedure is the second important feature of HBPSO/TS, and we used it to generate different solutions, and to lead the search to enter new hopeful areas. To assess whether the mutation procedure based on tabu is efficient, we carried out an experiment to contrast HBPSO/TS with two of its variants. We carried out an experiment to contrast HBPSO/TS with two of its variants in order to assess the validity of the tabu-based mutation procedure. To be exact, HBPSO/TS(UT) and HBPSO/TS(UM) were used to represent HBPSO/TS free of tabu-based mutation procedure and HBPSO/TS with $\lambda=0$ (i.e., all variables flip freely at all generations), respectively. We ran HBPSO/TS, HBPSO/TS(UM), and HBPSO/TS(UT) on three instances: sukp 500_485_0.15_0.85 (Fi10), sukp 500_500_0.15_0.85 (Si10), and sukp 485_500_0.15_0.85 (Ti10). Each was run 40 times. Fig. 7 plots the average objective function values obtained by HBPSO/TS, HBPSO/TS(UM) and HBPSO/TS(UT) on the three instances. From Fig. 7, one can see that HBPSO/TS outperforms HBPSO/TS(UM) and HBPSO/TS(UT), and HBPSO/TS(UM) performs better than HBPSO/TS(UT). This shows that the tabu based mutation procedure helps the search to explore new promising regions.

As described in Section 4.2, HBPSO/TS used the tabu search procedure to intensify the search. To assess the impact of the tabu search procedure on the HBPSO/TS performance, we carried out an experiment to contrast HBPSO/TS with two of its variants (HBPSO/TS(UL) and HBPSO/NLS). HBPSO/TS(UL) is HBPSO/TS without tabu search procedure, and HBPSO/NLS replaces the proposed tabu search with the neighborhood local search, which is used in (Chih, 2018). HBPSO/TS(UL) and HBPSO/NLS were operated 40 times on the three instances: Fi10, Si10, and Ti10 (defined

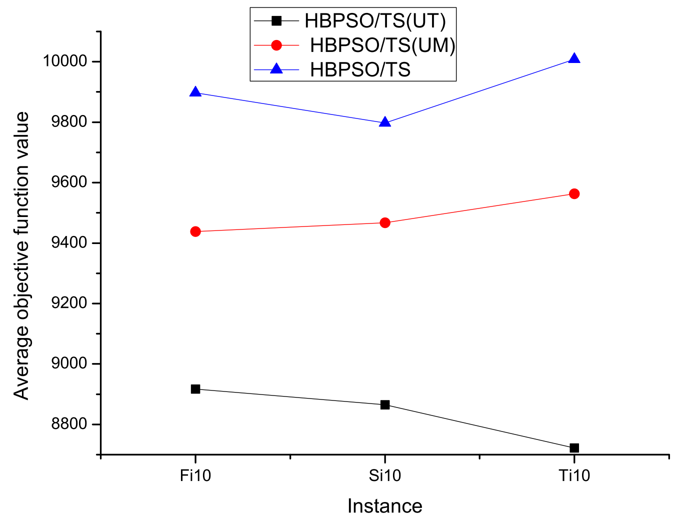

Fig. 7. The average objective function values obtained by HBPSO/TS, HBPSO/TS(UM) and HBPSO/TS(UT).

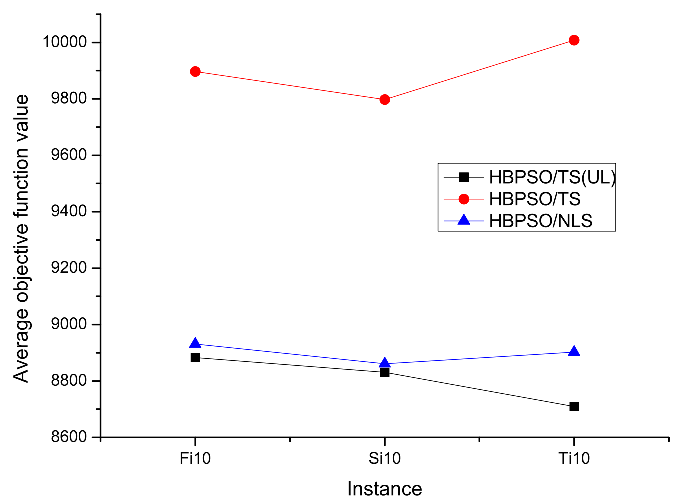

Fig. 8. The average objective function values obtained by HBPSO/TS, and HBPSO/TS(UL).

above). Fig. 8 plots the average objective function values obtained by HBPSO/TS and HBPSO/TS(UL) on the three instances. As we can observe from Fig. 8, much better mean objective function values can be found in HBPSO/TS than HBPSO/TS(UL) and HBPSO/NLS, which illustrates that the proposed tabu search procedure significantly improved the performance of the HBPSO/TS.

5. Conclusions

We have proposed a hybrid BPSO with tabu search for solving the set-union knapsack problem. The HBPSO/TS used an adaptive penalty function to evaluate the quality of solutions, and to ensure the feasibility of the search. According to the specific information of SUKP, we redefined the position updating rule to procedure new solutions, and designed a tabu based mutation procedure to diversify the search. In addition, a tabu search procedure was developed to intensify the search. Experiments were performed on three sets of benchmark instances. Results show that HBPSO/TS outperformed A-SUKP, GA, BABC, ABC_*bin*, and binDE with regards to solution quality. However, the computational cost of HBPSO/TS was larger than those of the comparison algorithms.

The impacts of three essential HBPSO/TS components were also investigated. We carried out experiments to demonstrate that

HBPSO/TS performs better than HBPSO/TS with exact penalty function, and showed that the tabu based mutation procedure and the tabu search procedure are essential to enhance the computational efficiency of HBPSO/TS.

Future research will be focused in the application of this methodology in related combinatorial optimization problems.

Credit Author Statement

Geng Lin and Jian Guan designed the proposed algorithm, Zuoyong Li and Huibin Feng performed the experiments, Geng Lin and Zuoyong Li wrote the manuscript, Geng Lin conceived the idea and contributed to the revision of the manuscript.

Declaration of Competing Interest

The authors declared that they have no conflicts of interest to this work. We declare that we do not have any commercial or associative interest that represents a conflict of interest in connection with the work submitted.

Acknowledgments

This research was supported partially by the National Natural Science Foundation of China under Grant 11301255, Fund by Collaborative Innovation Center of IoT Industrialization and Intelligent Production, Minjiang University (IIC1703), and the Program for New Century Excellent Talents in Fujian Province University.

References

Ali, M. M., & Zhu, W. (2013). A penalty function-based differential evolution algorithm for constrained global optimization. Computational Optimization and Applications, 54(3), 707–739.
Arulselvan, A. (2014). A note on the set union knapsack porblem. Discrete Applied Mathematics, 169, 214–218.
Balaji, S., & Revathi, N. (2016). A new approach for solving set covering problem using jumping particle swarm optimization method. Natural Computing, 15, 503–517.
Changdar, C., Mahapatra, G. S., & Pal, R. K. (2015). An improved genetic algorithm based approach to solve constrained knapsack problem in fuzzy environment. Expert Systems with Applications, 42(4), 2276–2286.
Chen, Y., & Hao, J. K. (2016). Memetic search for the generalized quadratic multiple knapsack problem. IEEE Transactions on Evolutionary Computation, 20(6), 908–923.
Chen, Y., Hao, J. K., & Glover, F. (2016). An evolutionary path relinking approach for the quadratic multiple knapsack problem. *Knowledge-Based Systems, 92*, 23–34.
Chih, M. (2018). Three pseudo-utility ratio-inspired particle swarm optimization with local search for multidimensional knapsack problem. Swarm and Evolutionary Computation, 39, 279–296.
Chuang, L. Y., Yang, C. H., & Li, J. C. (2011). Chaotic maps based on binary particle swarm optimization for feature selection. Applied Soft Computing, 11, 239–248.
Dai, J., Han, H., Hu, Q., & Liu, M. (2016). Discrete particle swarm optimization approach for cost sensitive attribute reduction. Knowledge-Based Systems, 102, 116–126.
Derrac, J., García, S., Molina, D., & Herrera, F. (2011). A practical turorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. Swarm and Evolutionary Computation, 1(1), 3–18.
Engelbrecht, A. P., & Pampara, G. (2007). Binary differential evolution strategies. In IEEE congress on evolutionary computation (pp. 1942–1947). IEEE.
García, F. J. M., & Pérez, J. A. M. (2008). Jumping frogs optimization: a new swarm method for discrete optimization. Technical report deioc 3/2008 Department of Statistics, O.R and computing, University of La Laguna, Tenerife, Spain.
García, S., Molina, D., Lozano, M., & Herrera, F. (2009). A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: A case study on the CEC'2005 special session on real parameter optimization. Journal of Heuristics, 15(6), 617–644.
Goldschmidt, O., Nehme, D., & Yu, G. (1994). Note: On the set-union knapsack porblem. Naval Research Logistic, 41(6), 833–842.
Gong, M. G., Yan, J., Shen, B., Ma, L., & Cai, Q. (2016). Influence maximization in social networks based on discrete particle swarm optimization. Information Sciences, 367–368, 600–614.
Haddar, B., Khemakhem, M., Hanafi, S., & Wilbaut, C. (2015). A hybrid heuristic for the 0–1 knapsack sharing problem. Expert Systems with Applications, 42, 4653–4666.
Haddar, B., Khemakhem, M., Hanafi, S., & Wilbaut, C. (2016). A hybrid quantum particle swarm optimization for the multidimensional knapsack problem. Engineering Applications of Artificial Intelligence, 55, 1–13.
He, Y., Wang, X., He, Y., Zhao, S., & Li, W. (2016a). Exact and approximate algorithms for discounted $\{0-1\}$ knapsack problem. Information Sciences, 369, 634–647.
He, Y., Xie, H., Wong, T., & Wang, X. (2018). A novel binary artificial bee colony algorithm for the set-union knapsack problem. *Future Generation Computer Systems, 78*, 77–86.
He, Y., Zhang, X., Li, W., Wu, W., & Gao, S. (2016b). Algorithms for the randomized time-varying knapsack problems. Journal of Combinatorial Optimization, 31(1), 95–117.
Jain, I., Jain, V. K., & Jain, R. (2018). Correlation feature selection based improved-binary particle swarm optimization for gene selection and cancer classification. Applied Soft Computing, 62, 203–215.
Jiang, F., Xia, H., Tran, Q. A., Ha, Q. M., Tran, N. Q., & Hu, J. (2017). A new binary hybrid particle swarm optimization with wavelet mutation. Knowledge-Based Systems, 130, 90–101.
Kellerer, H., Pferschy, U., & Pisinger, D. (2004). Knapsack problems. Berlin: Springer.
Kennedy, J., & Eberhart, R. (1995). Particle swarm optimization. Proceedings of the IEEE International Conference on Neural Networks, 4, 1942–1948.
Kennedy, J., & Eberhart, R. (1997). A discrete binary version of the particle swarm algorithm. In Proceedings of the IEEE international conference on computational cybernetics and simulation (pp. 4104–4108). Volume 5.
Kiran, M. S. (2015). The continuous artificial bee colony algorithm for binary optimization. Applied Soft Computing, 33, 15–23.
Lin, G., & Guan, J. (2018a). A hybrid binary particle swarm optimization for the obnoxious p-median problem. Information Sciences, 425, 1–17.
Lin, G., & Guan, J. (2018b). Solving maximum set *k*-covering problem by an adaptive binary particle swarm optimization method. Knowledge-Based Systems, 142, 95–107.
Lin, G., Zhu, W., & Ali, M. M. (2016). An effective hybrid memetic algorithm for the minimum weight dominating set problem. *IEEE Transactions on Evolutionary Computation, 20*(6), 892–907.
Lin, G., & Zhu, W. X. (2014). An efficient memetic algorithm for the max-bisection problem. IEEE Transactions on Computers, 63(6), 1365–1376.
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.
Mahi, M., Baykan, O. K., & Kodaz, H. (2018). A new approach based on particle swarm optimization algorithm for solving data allocation problem. Applied Soft Computing, 62, 571–578.
Marinakis, Y., Migdalas, A., & Sifaleras, A. (2017). A hybrid particle swarm optimization - variable neighborhood search algorithm for constrained shortest path problems. European Journal of Operational Research, 261(3), 819–834.
Meng, T., & Pan, Q. K. (2017). An improved fruit fly optimization algorithm for solving the multidimensional knapsack problem. Applied Soft Computing, 50, 79–93.
Navathe, S., Ceri, S., Wiederhold, G., & Dou, J. (1984). Vertical partitioning algorithms for database design. ACM Transactions on Database Systems, 9, 680–710.
Pisinger, D. (1995). An expanding-core algorithm for the exact 0–1 knapsack problem. European Journal of Operational Research, 87, 175–187.
Schmitt, L. M. (2001). Theory of genetic algorithms. Theoretical Computer Science, 259(1–2), 1–61.
Tang, C. S., & Denardo, E. V. (1988). Models arising from a flexible manufacturing machine, part II: Minimization of the number of switching instants. Operations Research, 36(5), 778–784.
Tu, M., & Xiao, L. (2016). System resilience enhancement through modularization for large scale cyber systems. In In 2016 IEEE/CIC international conference on comunications in china. IEEE.
Wang, H., Sun, H., Li, C., Rahnamayan, S., & Pan, J. S. (2013). Diversity enhanced particle swarm optimization with neighborhood search. Information Sciences, 223, 119–135.
Wu, Q., & Hao, J. K. (2013). Memetic search for the max-bisection problem. Computers & Operations Research, 40(1), 166–179.