Finding and Exploring Promising Search Space for the 0-1 Multidimensional Knapsack Problem

Jitao Xu, Hongbo Li, Minghao Yin
College of Information Science and Technology, Northeast Normal University, Changchun, China.

Abstract
The 0-1 Multidimensional Knapsack Problem (MKP) is a classical NP-hard combinatorial optimization problem with many engineering applications. In this paper, we propose a novel algorithm combining evolutionary computation with the exact algorithm to solve the 0-1 MKP. It maintains a set of solutions and utilizes the information from the population to extract good partial assignments. To find high-quality solutions, an exact algorithm is applied to explore the promising search space specified by the good partial assignments. The new solutions are used to update the population. Thus, the good partial assignments evolve towards a better direction with the improvement of the population. Extensive experimentation with commonly used benchmark sets shows that our algorithm outperforms the state-of-the-art heuristic algorithms, TPTEA and DQPSO, as well as the commercial solver CPLEX. It finds better solutions than the existing algorithms and provides new lower bounds for 10 large and hard instances.

Keywords: 0-1 Multidimensional Knapsack Problem, Evolutionary Computation, Heuristic, Exact Algorithm, Large Neighbourhood Search

1. Introduction
The 0-1 Multidimensional Knapsack Problem (MKP) is a well-known combinatorial optimization problem that has been applied in different practical domains. Given a set \( N = \{1, 2, ..., n\} \) of items with profits \( p_i > 0 \) and a set \( M = \{1, 2, ..., m\} \) of resources with a capacity \( b_j > 0 \) for each resource. Each item \( i \) consumes a given amount of each resource \( r_{ij} > 0 \). The 0-1 MKP is to select a subset of items that maximize the sum of the profits without exceeding the capacity of each resource. The problem is formally stated as:

\[
\text{Maximize } f = \sum_{i=1}^{n} p_i x_i \\
\text{s.t. } \sum_{i=1}^{n} r_{ij} x_i \leq b_j, j \in M = \{1, 2, ..., m\} \\
x_i \in \{0, 1\}, i \in N = \{1, 2, ..., n\}
\]

where each \( x_i \) is a binary variable indicating whether the item \( i \) is selected, i.e., \( x_i = 1 \) if the item is selected, and 0 otherwise.

The 0-1 MKP has many real-world engineering applications, such as resource allocation \cite{11, 12, 13}, cutting stock \cite{4}, portfolio-selection \cite{5}, obnoxious facility location \cite{6}, slice admission control problem \cite{7}, food order optimization problems \cite{8}, and multi-unit combinatorial auction \cite{9}. Due to its NP-hard characteristic, solving the MKP is computationally challenging. The exact algorithms for the MKP are usually based on branch and bound (B&B) algorithms, such as the CORAL algorithm \cite{10}. However, the modern exact algorithms could solve only the MKP instances of relatively small and moderate size, i.e., the instances with less than 250 items and less than 10 resource constraints. For larger instances with more than 500 items and more than 30 resource constraints, the heuristic algorithms are usually better choices.

There exist a number of heuristic algorithms for the problem. For instance, the single-solution based local search algorithms maintain a solution during the search process and iteratively optimize it. The representative of these algorithms include Tabu Search \cite{11}, Simulated Annealing \cite{12}, Kernel Search \cite{13}, and Core Problem Algorithm \cite{14}. In recent years, the population-based evolutionary algorithms are more popular for the problem, such as Particle Swarm Optimization (PSO) \cite{15, 16, 17, 18, 19, 20}, Ant Colony Optimization \cite{21, 22}, Moth Search Optimization \cite{23}, Pigeon Inspired Optimization \cite{24, 25}, Bee Colony Algorithm \cite{26}, Quantum Cuckoo Search Algorithm \cite{27}, Fruit Fly Optimization Algorithm \cite{28, 29}, and Grey Wolf Optimizer \cite{30}. Chih et al. \cite{15}
introduced two algorithms based on binary PSO to address the MKP. Building upon this research, Chih \[16\] further improved the efficacy of PSO in handling the MKP by incorporating the self-adaptive check and repair operator (SACRO) mechanism. This enhancement was subsequently refined in 2018 by introducing an advanced SACRO concept integrating three pseudo-utility ratios to augment efficiency \[17\]. In a different approach, Haddar et al. \[18\] adopts quantum particle swarm optimization (QPSO) to address the MKP, while Lai et al. \[19\] innovated a diversity-preserving mechanism aimed at enhancing the diversity of the population in QPSO. Hybrid algorithms have also been proposed, such as the NP+BAS+LP algorithm introduced by Al-Shihabi and Olafsson \[21\]. This hybrid approach integrates Binary Ant System (BAS) within a Nested Partition (NP) framework, leveraging Linear Programming (LP) relaxations to guide partitioning and improve solution quality. BAS facilitates the generation of feasible solutions within each partition, thereby enhancing search efficiency. On the nature-inspired optimization front, Feng and Wang \[23\] proposed a binary moth search algorithm specifically designed for the MKP. Additionally, Setiawan et al. \[24\] introduced Pigeon-Inspired Optimization (PIO) for solving the MKP. Further contributions include an enhanced binary heuristic algorithm based on the Artificial Bee Colony (ABC) method by Wei \[25\]. Binary Fruit Fly Optimization Algorithm (bFOA) proposed by Wang et al. \[29\], and an enhanced version of Fruit Fly Optimization Algorithm (IFFOA) by Meng and Pan \[28\]. Each of these algorithms brings unique strategies to the table, such as utilizing binary strings to represent solutions, incorporating various search processes, and employing repair operators to ensure solution feasibility. Moreover, Luo and Zhao \[30\] utilized The Binary Grey Wolf Optimizer (GWO) for solving the MKP. This approach combines critical elements, including an initial elite population. (3) Our algorithm provides new a lower bound for 8 hard and large instances.

The paper is organized as follows. We first introduce the motivation and the raw idea of our algorithm in Section 2. The details of the proposed algorithm, including the framework of the new algorithm, and the adaption of the TPTEA algorithm for finding good partial assignments, a customized LNS, are present in Section 3. The experimental results are shown in Section 4. We discussed our algorithm with some closely related methods in Section 5. Finally, Section 6 is the conclusion.

2. The Motivation

An assignment $A$ to items $I \subseteq N$ is a set of instantiations of the form $(x_i = v_i)$, one for each $i \in N$ to assign $v_i$ to $x_i$ where $v_i \in \{0, 1\}$. If $I = N$, then $A$ is a complete assignment; otherwise a partial assignment. A complete assignment $s$ that satisfies all resource constraints is a feasible solutions to the instance, or a solution for short. A solution $s$ is an optimal solution if its objective value $f(s)$ is larger than or equal to that of any other solution to the instance. A partial assignment is an optimal one if it can be extended to an optimal solution.
Given an MKP instance $P$ with $n$ items, the size of the search space of finding an optimal solution for $P$ is $2^n$. Given a partial assignment $A$ of $P$. Fixing the values of $A$ in $P$ will result in a sub-problem $P'$ whose search space is $2^n-|A|$. The search space of $P'$ can be considered as a sub search space of $P$ which is specified by $A$. If we are given an optimal partial assignment $A^*$ of $P$, then we can find an optimal solution for $P$ in a search space of size $2^n-|A^*|$ instead of the entire search space of size $2^n$. However, as far as we know, there is no existing method that identifies optimal partial assignment for general 0-1 MKP before solving it exactly. Thus, the aim of our algorithm is to find good partial assignments that have a large possibility of being an optimal one, or can be extended to a high-quality solution whose objective value is close to that of an optimal solution. Then exploring the promising search space specified by the good partial assignments may find high-quality solutions efficiently.

Evolutionary Algorithms (EA) are powerful techniques for solving combinatorial optimization problems based on the simulation of biological evolution. They start from a population of individuals (representing solutions) and produces offspring (new solutions) by utilizing the information of the population. Guided by a fitness function measuring the quality of a solution, the new solutions with high-quality are usually used to update the population. As a result, the population evolves towards the direction of better fitness function. One of the advantages of EA is to identify and keep some high-quality patterns (partial assignments) in the population, so they can produce high-quality solutions through crossover and mutation operations. Thus, we propose a novel algorithm that combines Evolutionary Algorithm with exact algorithm for the 0-1 MKP. The main idea of the algorithm is shown in Algorithm 1.

Algorithm 1: The main idea

| Input: | a MKP instance $P$; |
|------------------|------------------|
| Output: | current best solution $s^*$; |
| $s^*$ ← | the best solution in population; |
| while | the termination condition is not reached do |
| $pa$ ← | extract(population); |
| $s$ ← | explore($P, pa$); |
| updatePopulation($s$); |
| if $f(s) > f(s^*)$ then |
| $s^*$ ← | $s$; |
| return $s^*$; |

At the beginning, the population of Evolutionary Algorithm, a set of solutions recorded in $population$ is initialized at line 1. The $s^*$ records the best solution found so far. The loop at lines 3-8 is the main procedure, e.g., a promising partial assignment $pa$ is extracted from the population by an Evolutionary Algorithm at line 4 and the promising search space is explored by an exact algorithm at line 5. The new solution is used to update the population at line 6. If a better solution is found, $s^*$ will be updated accordingly at lines 7-8. The procedure repeats until a termination condition is reached and the best found solution is returned. There are various termination conditions that can be used in the algorithm, such as a time limit or the number of iterations.

3. Related Work

There exist some methods combining Evolutionary Computation with exact algorithms for solving combinatorial optimization problems. We refer those ones related to our method in the following, and then discuss the differences at the end of next section.

Kernel Search Angelelli et al. [13] can be applied in solving the MKP. It divides the item set into two parts, the kernel, and the buckets, and then applies an exact algorithm in the kernel. Kernel Search finds the solution of the continuous relaxation problem, and sorts the items according to the information provided by the continuous solution. Then it selects the first $C$ items as a kernel $\Lambda$. The items belonging to $N \setminus \Lambda$ construct a sequence $\{B_i\}$ of buckets. Finally, an exact algorithm is iteratively applied in the kernel. In each iteration, some of the items in buckets are selected and added into the kernel. There are two additional constraints in each run of the exact algorithm: (1) the lower bound is the current best solution; (2) at least one of the latest added items should be selected in the solution.

The Generate And Solve framework (GAS) [37, 38] was originally designed for solving the container loading problem. The framework suggests decomposing the original problem into several sub-instances whose solutions are also the solutions of the original one, and then applying some exact algorithms in the sub-instances. Besides the GAS framework, the Construct, Merge, Solve & Adapt algorithm (CMSA) [39] suggests to generates sub-instances by merging different solution components found in probabilistically constructed solutions. The algorithm has been applied in solving the minimum common string partition (MCSP) problem, and a minimum covering arborescence (MCA) problem.
Gallardo et al. [40] propose a hybrid algorithm that combines exact algorithm with evolutionary computation for the MKP. It runs an evolutionary algorithm and an exact algorithm alternately. The two algorithm components share a single solution. Whenever one algorithm component A finds a new solution better than the current best solution, the other algorithm component B takes over and the new solution is used to assist algorithm component B. More specifically, when the exact algorithm finds a better solution, the solution is used to replace the worst one in the population of the evolutionary computation. On the other hand, when the evolutionary algorithm finds a better solution, the objective of the solution is used to update the lower bound of the exact algorithm.

4. Combining Evolutionary Algorithm with Integer Programming for the MKP

Based on the idea proposed in last section, we introduce the detailed algorithm for solving the 0-1 MKP, namely finding and exploring promising search space (FEPSS). The framework of the algorithm is presented first.

4.1. The Framework

The main procedure of FEPSS is present in Algorithm 2. At the beginning, the population, a set of solutions, is initialized at line 1 and the current best solution is initialized to the best one of the initial population. There are two cases to find and explore good partial assignments. In the first case (lines 10-12), the good partial assignments are extracted from the population at line 11 and the search space specified by $pa$ is explored at line 12. In the second case (lines 6-8), we run a customized Large Neighbourhood Search procedure to refine current best solution. Each of the two cases returns a new solution $s$, which is used to update the population at line 13 and update the current best solution at lines 14-16. A counter $num$ is used to record the number of the first case is executed. If the counter reaches a predefined parameter $lnsLimit$ or the best solution is updated, the second case will be executed once. The details of the procedures called in the framework will be introduced in the following subsections.

Algorithm 2: The framework

```
Input: a MKP instance $P$;
Output: current best solution $s^*$;

1 population ← initPopulation($P$);
2 $s^*$ ← the best solution in population;
3 num ← 0;
4 while the termination condition is not reached do
5     if iterationNum = lnsLimit or bestUpdated then
6         bestUpdated ← false;
7         num ← 0;
8         s ← LNS($P$, $s^*$);
9     else
10        num ← num + 1;
11        $pa$ ← extractFromPopulation($P$);
12        s ← explore($P$, $pa$);
13        updatePopulation(s);
14     if $f(s) > f(s^*)$ then
15         bestUpdated ← true;
16         $s^*$ ← s;
17 return $s^*$;
```

Note that there is a preprocessing step ahead of the algorithm, e.g., all the items are preprocessed and renumbered in an ascending order of the surrogate relaxation ratio [41] evaluation of the items, which is defined as follows.

$$c_i = \frac{p_i}{\sum_{j=1}^{n} \frac{r_{ij}}{b_j}}, \forall i \in \{1, 2, ..., n\}$$  \hspace{1cm} (1)

After the items are renumbered, the corresponding $p_i$ and $r_{ij}$ are adjusted accordingly. In the following, all the items will be used with their new numbering. An item $i$ is selected (not selected) in a solution means that the value of the corresponding variable $x_i$ is 1 (0), and vice versa. $population[k]$ is the $k$-th solution in population and $s[z]$ is the value of variable $x_{i}$ in solution $s$.

In the following subsections, we will introduce the initPopulation, extractFromPopulation, LNS, explore and updatePopulation procedures in details.
4.2. Initializing the Population

The solution set population contains ns solutions. At the beginning, we randomly generate ns × 1000 solutions and add the best ns ones into population (selected by their objectives). Then we run a tabu search to refine the random solutions in population. The procedure of generating initial population is present in Algorithm 3.

When generating a random solution, the geneRandomSolutions procedure randomly selects an item and tries to add it into the knapsack. The item is added into the knapsack if no resource constraint is violated; otherwise, it is skipped. A random solution will be generated after all items are tried.

To refine the random solutions, the tabu search strategy of the first phase of TPTEA algorithm is employed [35]. We briefly recall the procedure here. It explores only the feasible search space and uses two basic neighborhoods: the one-flip neighborhood $N_1(s)$ and the swap neighborhood $N_2(s)$. The $N_1(s)$ contains all the feasible solutions which can be obtained by applying the flip operator that changes the value of the variable $x_q$ in $s$ to its complementary value $1-x_q$. The $N_2(s)$ contains all the feasible solutions which can be obtained by applying the swap operator that swaps the values of two variables with different values in $s$. The tabuSearch procedure is shown in Algorithm 4. It always finds the best neighbour from current neighbourhood $N_1(s) \cup N_2(s)$ and mains a record of all the visited solutions to avoid duplicate visiting. The tabu function is implemented with three hashing vectors [42]. The best solution visited during the tabu search is returned.

Algorithm 3: initPopulation

Input: a MKP instance $P$; 
Output: a solution set population;

1. population ← geneRandomSolutions();
2. for $i = 1$ to ns do
   3.  population[$i$] ← tabuSearch(population[$i$]);
4. return population;

Algorithm 4: tabuSearch

Input: a solution $s$; 
Output: a refined solution $s'$;

1. iteration ← 1;
2. tabuList ← ∅;
3. $s' \leftarrow s$;
4. while iteration $\leq$ maxIteration do
   5.  find a best neighbor solution $s_{neighbor}$ from $N_1(s) \cup N_2(s)$, which is not in tabuList;
   6.  $s \leftarrow s_{neighbor}$;
   7.  if $f(s) > f(s')$ then
      8.    $s' \leftarrow s$;
   9.  add $s$ into tabuList;
   10. iteration ← iteration + 1;
5. return $s'$;

4.3. Extracting High-quality Partial Assignments from the Population

In this subsection, we introduce how to extract good partial assignments from the population. The idea is to divide the items into two parts: the fixed part and the free part. Then we generate a partial assignment for the fixed part and explore the free part with an exact algorithm. The extractFromPopulation procedure is shown in Algorithm 5. It decides items in the fixed part at lines 1-8 and then assigns values to the items in the fixed part at lines 13-25 to generate a partial assignment. The details of the algorithm are as follows.

Firstly, each item $i$ is associated with a score $\delta_i$ defined as:

$$\delta_i = \beta \times \frac{w_i}{ns} + (1 - \beta) \times \frac{1}{2} + \frac{rand(0, 1)}{2}$$

(2)

where $\beta \in (0, 1)$ is a parameter, $rand(0, 1)$ returns a pseudo random number between (0, 1) and $w_i$ is defined as:

$$w_i = \sum_{k=1}^{ns} \text{population}[k][i]$$

(3)
Algorithm 5: extractFromPopulation

Input: a MKP instance $P$;
Output: a partial assignment $pa$;
1 calculate $\delta_i$ for each item $i$;
2 $t \leftarrow$ number of items selected in $s^*$;
3 sort all items by the ascending order of $\delta_i$;
4 $fix \leftarrow \emptyset$;
5 for each $i \in [1, n - t - \Delta)$ do
6     $fix \leftarrow fix \cup \{i\}$;
7 for each $i \in (n - t + \Delta, n]$ do
8     $fix \leftarrow fix \cup \{i\}$;
9 $num \leftarrow \frac{ns}{10}$;
10 $pa \leftarrow \emptyset$;
11 exceed $\leftarrow$ true;
12 while exceed do
13     for each $i \in fix$ do
14         selectedNum $\leftarrow$ 0;
15         for $j = 1$ to $num$ do
16             $s \leftarrow$ select a solution from $population$ randomly;
17             if $s[i] = 1$ then
18                 selectedNum $\leftarrow$ selectedNum $+$ 1;
19         $v \leftarrow$ 1;
20         if selectedNum $< \frac{num}{2}$ then
21             $v \leftarrow$ 0;
22         $r \leftarrow$ rand$(0, 1)$;
23         if $r < \frac{1}{n \times 10}$ then
24             $v \leftarrow 1 - v$;
25     $pa \leftarrow pa \cup (x, v)$;
26     if the items selected in $pa$ violate no resource constraint then
27         exceed $\leftarrow$ false;
28     else
29         $pa \leftarrow \emptyset$;
30 return $pa$;

Thus, $\frac{ns}{10}$ indicates the proportion of solutions in $population$ selecting item $i$. The items are renumbered by the score defined in formula (1), so $\frac{1}{n}$ indicates how likely item $i$ is selected by the surrogate relaxation ratio. We would prefer adding the items with higher $\delta_i$ into the knapsack.

Secondly, we count the number of items selected in current best solution $s^*$, i.e., $t = \sum_{i=1}^{n} s^*[i]$. Then we sort the items in ascending order of $\delta_i$. The items $i \in [1, n - t - \Delta)$ and the items $i \in (n - t + \Delta, n]$ compose a fixed part which is recorded in $fix$ at lines 4-8, and the remaining items $i \in [n - t - \Delta, n - t + \Delta]$ compose a free part, where $\Delta$ is a parameter. We make such selections because the items with smaller $\delta_i$ have larger possibility of not being selected and the ones with larger $\delta_i$ have larger possibility of being selected, and the free part should contain those hard to decide items. The strategy eliminates the items locating near the head (items with small $\delta_i$) and the end (items with large $\delta_i$) of the ordering. As is shown in Figure 1, the blue ones are fixed part and the red ones are free part.

![Figure 1: Example of fixed part and free part.](image)

Although the score $\delta_i$ gives a recommendation to the values of variables in the fixed part, we design a more elaborate strategy to determine the values to avoid local optimum (lines 14-21 in Algorithm 5). For each item in the
fixed part, we randomly select $\frac{na}{10}$ solutions from the solution set population. Then we count the number of the selected solutions containing the item, recorded in selectedNum (lines 15-18). An item is selected if more than half of the $\frac{na}{10}$ solutions contain the item (lines 19-21). A mutation operation that flips the value with probability $\frac{na}{12}$ is used at lines 22 to 24. The while loop at line 12 and the if condition at line 26 ensure that the generated partial assignment violates no resource constraint. After the partial assignment $pa$ is generated, an exact algorithm (Integer Programming) is employed to explore the corresponding search space, which will be introduced later.

4.4. The Customized Large Neighbourhood Search

In this subsection, we introduce a customized Large Neighbourhood Search procedure to refine current best solution. The procedure of extracts good partial assignments from current best solution and explores the sub-space specified by the partial assignments to find high-quality solutions. It also uses the information of the population to select some items that compose a free part and the remaining items compose the fixed part. We keep the values in current best solution for the fixed part to generate a partial assignment. The LNS procedure is shown in Algorithm

Algorithm 6: LNS

Input: a MKP $P$ and current best solution $s^*$
Output: a solution $s$
1 $s \leftarrow \text{null}$;
2 for $i = 1$ to lnsIterationNum do
3 \hspace{1em} var1 $\leftarrow$ votingFunction();
4 \hspace{1em} var2 $\leftarrow$ randomFunction();
5 \hspace{1em} var3 $\leftarrow$ ratioFunction();
6 \hspace{1em} free $\leftarrow$ var1 $\cup$ var2 $\cup$ var3;
7 \hspace{1em} pa $\leftarrow$ $s^*$ $\setminus$ free;
8 \hspace{1em} $s' \leftarrow$ explore($P$, $pa$);
9 \hspace{1em} updatePopulation($s'$);
10 \hspace{1em} if $s = \text{null}$ or $f(s') > f(s)$ then
11 \hspace{2em} $s \leftarrow s'$;
12 return $s$;

We propose three functions, votingFunction, randomFunction and ratioFunction to select the items belonging to the free part. Each of them will find a set of items and their union constitutes the free part (lines 3-7). Then we remove the assignments of the free part from current best solution $s^*$, and the remaining constitute a partial assignment $pa$ (line 7).

The votingFunction is based on the score $w_l$ defined in formula (3) which records the number of solutions in population voting for the selection of the item. The function selects items into free part in two cases: (1) item $i$ is in $s^*$ and $w_i < \text{ns} - \sigma$; (2) item $i$ is not in $s^*$ and $w_i > \sigma$, where $\sigma$ is the standard deviation of $W = \{w_1, w_2, ..., w_n\}$.

The following two functions work on the enumerated items which are sorted by the ascending order of formula (1). The randomFunction simply selects each item indexed from index1 to $n$ with probability $P_1$, where index1 points to the first item with $w_i \geq \frac{na}{2}$.

The ratioFunction uses another index index2 that points to the first item with $w_i \geq 1$. It makes selections in the items indexed from index2 to index1 only. For each item $i$, we calculates its average profit $\frac{w_i}{r_{ij}}$ in each resource constraint $j$. In each resource constraint, we can find a set of items containing the top $t_2$ items with the largest average profit in this constraint. If an item has appeared in any of the sets, it will be selected with a probability $P_2$.

The LNS procedure calls the three functions to select the items belonging to the free part at lines 3-5. After that, it combines the three item sets to get the free part at line 6. Then it generates a partial assignment $pa$ by removing the assignments of the free part from current best solution $s^*$ at line 7. Finally, it calls the explore procedure to explore the search space specified by the partial assignment at line 8. The procedure repeats lnsIterationNum times and the best solution found in the LNS procedure is returned.

4.5. Exploring A Promising Search Space

In explore procedure shown in Algorithm we add all the items selected by the partial assignment into the knapsack and calculate the remaining resource for each resource constraint $j$, $b'_j = b_j - \sum_{i \in \text{fixed part}} r_{ij}x_i$. Then we calculate the total cost, $t_{c_j} = \sum_{i \in \text{free part}} r_{ij}$, of each resource $j$ of all the free items. The resource constraint with largest deficit is the one with the largest $\frac{t_{c_j}}{b'_j}$, marked by $l$ at line 8. We calculate $\frac{t_{c_j}}{b'_j}$ for each item $i$ involved in $pa$ and sort the items by the descending order of $\frac{t_{c_j}}{b'_j}$. Then the assignment of top $t_1$ items are removed from $pa$, i.e., added into free part. Finally, we fix the values of the items that are still in $pa$ and run Integer Programming to solve the sub problem within a given time limit $T$ seconds, i.e., the IP may not complete. The best solution found within $T$ seconds is returned.
Algorithm 7: explore

Input: a MKP instance $P$ and the fixed part $pa$;
Output: a solution $s$;
1 for each resource constraint $j$ do
2 \hspace{1em} $b'_j \leftarrow b_j$;
3 \hspace{1em} for each $(x_i, v) \in pa$ do
4 \hspace{2em} $b'_j \leftarrow b'_j - r_{ij} \times v$;
5 \hspace{1em} $tc_j \leftarrow 0$;
6 \hspace{1em} for each item $i$ not involved in $pa$ do
7 \hspace{2em} $tc_j \leftarrow tc_j + r_{ij}$;
8 \hspace{1em} $l \leftarrow$ the index of the constraint with largest $\frac{tc_j}{b'_j}$;
9 \hspace{1em} calculate $\frac{b_i}{b_j}$ for each item involved in $pa$;
10 \hspace{1em} sort the items by the descending order of $\frac{b_i}{b_j}$ and remove the assignment of the top $t_1$ items from $pa$;
11 \hspace{1em} for each $(x_i, v) \in pa$ do
12 \hspace{2em} fix the value of $x_i$ to $v$;
13 \hspace{1em} $s \leftarrow$ run Integer Programming to solve the sub problem with the fixed values in $T$ seconds;
14 return $s$;

4.6. Updating the Population

When updating the population with a new solution $s$, the updatePopulation procedure adds $s$ into population, and removes the one with the smallest scoring function $V(s)$ defined as:

$$V(s) = \alpha \times \frac{f(s) - f_{\text{min}}}{f_{\text{max}} - f_{\text{min}}} + \frac{1 - \alpha}{2} \times \left( \frac{D(s) - D_{\text{min}}}{D_{\text{max}} - D_{\text{min}}} + \frac{R(s) - R_{\text{min}}}{R_{\text{max}} - R_{\text{min}}} \right)$$ (4)

$$D(s) = \min \{ \text{distance}(s, s') : s, s' \in \text{population}, s \neq s' \}$$ (5)

$$R(s) = \sum_{i=0}^{n} c_i \times s[i]$$ (6)

where $\alpha \in [0 - 1]$ is a parameter, distance is the Hamming distance, $c_i$ is defined in formula (1), the min and max denote the minimum and maximum ones of the corresponding scores respectively. We always keep current best solution in population. If current best solution has the smallest $V(s)$, then we remove the one with the second smallest $V(s)$. Note that if $s$ is already in population, the population will not updated.

4.7. Discussions

Our algorithm can be considered as the combination of Evolutionary Computation \cite{43} and Large Neighbourhood Search \cite{36}. It maintains a set of solutions, extracts information from the population to help find high-quality solutions, and updates the population when new solutions are found. All of these operations are within the framework of Evolutionary Computation and are customized for the 0-1 MKP in our algorithm. The idea of LNS is to find a fixed part from the current best solution and explore the remaining search space. The extractFromPopulation procedure is simulating LNS. The major difference is that the fixed part is generated from the population, not from the current best solution. Theoretically, the fixed part extracted by extractFromPopulation procedure may exceed some resource constraints, but we did not observe this case in our experiments. On the other hand, the fixed part generated from the current best solution will not exceed any resource constraint. We discuss the differences between FEPSS and the related methods in the following.

The main idea of Kernel Search is similar to that of FEPSS with the kernel corresponding to the free part of FEPSS and the buckets corresponding to the fixed part. However, the procedure of finding the kernel is different from the procedure of finding the free part in FEPSS. In addition, the fixed part of FEPSS contains both selected items and non-selected ones, while the buckets contain only non-selected items.

Our algorithm can be considered as an instantiation of the GAS framework \cite{37,38}. While the GAS framework suggests decomposing the original problem into several sub-instances, our method employs the idea of evolutionary computation to generate the fixed part (some of the variables are fixed) which can be considered as a special case of
sub-instances (the domain of some variables are reduced). The major difference between CMSA and FEPSS is also the way of generating sub-instances. In addition, we propose a new scoring function for updating the solution set with the new solutions found by Integer Programming.

The hybrid algorithm \cite{40} runs an evolutionary algorithm and an exact algorithm alternately, so it is different from FEPSS, since FEPSS utilizes an evolutionary algorithm to find promising search spaces and runs an exact algorithm to explore the promising search spaces.

5. Experiments and Analysis

To examine the efficiency of FEPSS algorithm, finding high-quality solutions in a reasonable time, we performed vast experimentation with two benchmark sets that are widely used to test the algorithms for the MKP. Before presenting the results, we introduce some detailed settings of the experiments in the following.

**Benchmark.** We perform extensive computational experiments on the commonly used benchmark instances, the OR-Library instances, and the MK\_GK instances. The main characteristics of the benchmarks are as follows.

**OR-Library:** These instances were proposed in \cite{44}. In this set, the number of items \( n \) are 100, 250, and 500. The number of resource constraints \( m \) are 5, 10, and 30. Each combination \((n, m)\) consists of 30 instances. For each pair \((n, m)\), the coefficients \( b_i \) are set using \( b_i = \omega \sum_{j=1}^{n} r_{ij} \), where \( \omega \) represents a tightness ratio. Specifically, \( \omega = 0.25 \) for the first ten problems (instance ids from 0 to 9), \( \omega = 0.5 \) for the next ten problems (instance ids from 10 to 19), and \( \omega = 0.75 \) for the remaining ten problems (instance ids from 20 to 29). Here, \( r_{ij} \) are integer numbers drawn from a discrete uniform generator U(0, 1000). This is the most commonly used benchmark for the MKP. As far as we know, the optimal solutions for the small instances are proven and the \((250, 30), (500, 10), \) and \((500, 30)\) are still open. The instances are available at http://people.brunei.ac.uk/~mastjhb/jeb/orlib/mknapsinfo.html.

**MK\_GK:** There are 11 instances with \( n \in \{ 100, 200, 500, 1000, 1500, 2500 \} \) and \( m \in \{ 15, 25, 50, 100 \} \). This set contains some very large instances, e.g., the one with 2500 items and 100 resource constraints. The instances are made available by the authors of \cite{35} at https://leria-info.univ-angers.fr/~jinkao.hao/mkp.html.

**Baselines.** We compared our algorithm with the two state-of-the-art heuristic algorithms, the TPTEA algorithm and the DQPSO algorithm. Some of the best-known results are found by the algorithms combining linear programming with tabu search \cite{45,46}. The results were obtained by running the algorithms for several days, so we did not involve these algorithms in our experiments. We have used CPLEX as the Integer Programming solver. Although IP in FEPSS is allocated a short time (10 or 20 seconds) and we are not utilizing the full power of CPLEX, people are interested in the performance of CPLEX. Running CPLEX usually needs a large amount of space, so we include the best results of running CPLEX with 5Gb RAM for indicative purposes.

**Implementation Details.** We use the source codes of DQPSO provided by the authors and implemented a TPTEA algorithm because the source codes are not publicly available. Our algorithm is implemented in C++ and compiled using the g++ compiler with the -O3 option. We conducted experiments on Intel E7-4820 v4 (2.0GHz) running on a Linux system. The performance of the compared algorithms is measured by solution quality. We also include the CPU time of finding the best solution for indicative purposes.

**Metrics.** For each instance, we run each algorithm with 30 random seeds ranging from 1 to 30. Consequently, we obtain the best objective value \( f_{\text{best}} \) among the 30 runs, while \( f_{\text{worst}} \) represents the worst objective value among these 30 runs. Additionally, we calculate the average objective value \( f_{\text{avg}} \) and the average time cost \( t_{\text{avg}} \) of finding the best solution over the 30 runs. The best one in each comparison is highlighted in bold. Avg. denotes the average values of each column, and \#Best indicates the number of problems where the corresponding algorithm gets the best performance. The average time cost of finding the best solution in 30 runs is also presented in the following tables.

In the following, we first analyze how different components of FEPSS affect the efficiency, and then examine the parameters for FEPSS and find some appropriate values for the parameters. Finally, we compare FEPSS with the state-of-the-art heuristic algorithms for the MKP, TPTEA and DQPSO.

5.1. Ablation Analysis

We tested the impact of various components of FEPSS, including the customized large neighbourhood search (CLNS), the scoring function \( V(s) \), and the score of each item \( \delta_i \). The results are shown in Table 1. The different versions of FEPSS are introduced as follows:

**FEPSS-1:** This is the FEPSS without the customized large neighbourhood search. Specifically, we remove the lines 5 to 9 from Algorithm 2.

**FEPSS-2:** This is the version where the scoring function \( V(s) \) is removed. Without the scoring function, we randomly select and remove a solution from the population.

---

1The source code of our algorithm is available at https://github.com/jetou/FEPSS
Our results indicate that FEPPS achieves the best outcomes in terms of $f_{\text{best}}$ and $f_{\text{avg}}$. If we remove the CLNS component, FEPPS-1 saves a little time to find the current best solution and loses a little in solution quality. Moreover, when we replace the population update scoring function with a random strategy, we observe an increase in time consumption and the loss in solution quality, especially evident in the significant drop observed in $f_{\text{avg}}$. Additionally, FEPPS-3 analysis highlights the significant impact of each item’s score $\delta_i$ on the performance of FEPPS. FEPPS-3 loses in both solution quality and time costs. Based on these observations, we can infer that the effectiveness order of FEPPS components follows CLNS, the scoring function $V(s)$, and the score of each item $\delta_i$, in ascending order. The scoring function $\delta_i$ is effective in identifying the hard-to-decide items, so it finds promising search spaces containing high-quality solutions. These findings provide valuable insights into optimizing and refining the FEPPS algorithm for enhanced performance and efficiency.

5.2. Parameter Settings and Analysis

The OR-Library instances were generated with different tightness, so we examine the parameters in the representations of the instances with different tightness, e.g., 500-30-5 (tightness=0.25), 300-30-15 (tightness=0.5), and 500-30-25 (tightness=0.75). Each value of the parameters is tested with 30 random seeds from 1 to 30.

From the ablation study, we observe that the scoring function $\delta_i$ has a large impact on the performance of FEPPS, which indicates that the selection of the free part should further be investigated. Thus, we first examine the parameters affecting the selection of the free part, $\beta$ in formula (2) and $\Delta$ in Figure[1]. We tested $\beta$ ranging from 0.1 to 0.9, and present the results of the three instances in Figure[2]. The horizontal axis represents the running time, while the vertical axis denotes $f_{\text{avg}}$. The legends for the best three of $f_{\text{avg}}$ are highlighted in bold and the best one is in red. We can see that the best values of $\beta$ for the instances with tightness 0.25, 0.5, and 0.7 are 0.7, 0.6, and 0.5 respectively. The average of the best three $\beta$ values are 0.7, 0.6, and 0.47 respectively. The results indicate that with the increasing of the tightness, the best value for $\beta$ should be decreased. We can also observe that the value 0.6 appears in the best three values for all tightness. Thus, we use $\beta=0.6$ in the following experiments.

In Figure[3] we present the results of running FEPPS with different values of the parameter $\Delta$, which determines the length of the free part. We tested the values in $\{\frac{1}{10}, \frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}, \frac{9}{10}\}$. The horizontal axis represents the running time, while the vertical axis denotes $f_{\text{avg}}$. The legends corresponding to the top three $f_{\text{avg}}$ values are highlighted in bold and the best one is in red. In the figure, we observe a similar phenomenon as in Figure[2]. With the increasing of tightness, the best value for $\Delta$ decreases from $\frac{9}{10}$ to $\frac{3}{5}$, and the average of the best three values decrease from $\frac{4}{5}$ to $\frac{2}{5}$. Thus, we should run FEPPS with a smaller $\Delta$ for an MKP instance with larger tightness. The value $\frac{3}{5}$ appears in the best three values for all tightness. In the following experiments, we use $\Delta=\frac{3}{5}$.
(a) Parameter $\beta$ for 500-30-05 (tightness=0.25).

(b) Parameter $\beta$ for 500-30-15 (tightness=0.50).

(c) Parameter $\beta$ for 500-30-25 (tightness=0.75).

Figure 2: Influence of the parameter $\beta$ for the average objective value
(a) Parameter $\Delta$ for 500-30-05 (tightness=0.25).

(b) Parameter $\Delta$ for 500-30-15 (tightness=0.50).

(c) Parameter $\Delta$ for 500-30-25 (tightness=0.75).

Figure 3: Influence of the parameter $\Delta$ for the average objective value
Finally, we examined the parameter that determines the time limit for each run of the explore procedure. In other words, $T$ is the time limit of each run of the exact algorithm. We tested FEPPS with $T$ in $[5, 10, 30, 60, 300]$. Table 2 presents the results. The underlined results in the $f_{best}$ columns are those better than the best known results. We can see that when setting $T$ to 5, 10, 30, 60, and 300, FEPPS finds new solutions better than the best known results in 2, 3, 2, 3, and 0 instances. FEPPS with $T = 10, 30, 60$ find the best solution in 25 instances, which is the largest number. If we further check the average of the $f_{best}$, FEPPS with $T = 10$ outperforms all the others. When $T$ is set to 300, FEPPS performs poorly, so each run of the exact algorithm should not be allocated a large time limit. This is reasonable because the more time limit is allocated to one run of the exact algorithm, the less promising search spaces are explored. We also checked the ratio of time spent by the exact algorithm versus the exact algorithm in each run of FEPPS. The result is as we expected, e.g., more than 97% of time (more than 7200 seconds out of 7200 seconds) are spent by the exact algorithm and the evolutionary algorithm spends less than 3% of time. In general, FEPPS with $T = 10$ is suggested and we have found new lower bounds for 5 instances.

### Comparison with The Baselines

Finally, we examined the parameter that determines the time limit for each run of the explore procedure. In other words, $T$ is the time limit of each run of the exact algorithm. We tested FEPPS with $T$ in $[5, 10, 30, 60, 300]$. Table 2 presents the results. The underlined results in the $f_{best}$ columns are those better than the best known results. We can see that when setting $T$ to 5, 10, 30, 60, and 300, FEPPS finds new solutions better than the best known results in 2, 3, 2, 3, and 0 instances. FEPPS with $T = 10, 30, 60$ find the best solution in 25 instances, which is the largest number. If we further check the average of the $f_{best}$, FEPPS with $T = 10$ outperforms all the others. When $T$ is set to 300, FEPPS performs poorly, so each run of the exact algorithm should not be allocated a large time limit. This is reasonable because the more time limit is allocated to one run of the exact algorithm, the less promising search spaces are explored. We also checked the ratio of time spent by the exact algorithm versus the exact algorithm in each run of FEPPS. The result is as we expected, e.g., more than 97% of time (more than 7200 seconds out of 7200 seconds) are spent by the exact algorithm and the evolutionary algorithm spends less than 3% of time. In general, FEPPS with $T = 10$ is suggested and we have found new lower bounds for 5 instances.

All the parameters of FEPPS used in the experiments are listed in Table 3. Except for the three parameters examined above, the other parameters are arbitrarily set with intuition.

### 5.3. Comparison with The Baselines

Firstly, we present the average results of the OR-Library instances in Table 4. The results are the average of 30 instances of each combination ($n, m$). The OP/BK presents the optimal results or the best-known results. It is shown that FEPPS gets the best performance in the best objective comparison in 8 sets of instances, and gets the best performance in the worst objective comparison in 5 sets of instances, and gets the best performance in the worst objective comparison in 6 sets of instances. Note that it outperforms all the others in the three sets of large instances with 500 items. Although it requires more computational time than existing algorithms for 35 instances, e.g., item numbers 100 and 250, it consumes less computational time than TPTEA for larger instances, e.g., item number 500.

Secondly, the detailed results of the OR-Library instances (500-300). The BK column represents the best-known results of the instances. We include the BK column in the $f_{best}$ Comparison. It is shown that our BK finds the best-known results for 23 instances, while also establishing new lower bounds for 3 instances (the underlined results): 500.30.03, 500.30.04, and 500.30.18. In addition, From the $f_{avg}$ columns, we observe FEPPS achieves the best performance in 28 instances. Although $DQPSO$ costs less time than the other two, it is outperformed by the others in solution quality. The column $f_{worst}$ shows the worst case of the 30 runs of each instance. We can see that FEPPS achieves the best performance in all 30 instances in the $f_{worst}$ column. Furthermore, from the average of the standard deviations (Std column), we find that our algorithm has the lowest value, which indicates that our algorithm is more stable than the other algorithms. The small p-values ($< 0.05$)
**Table 3: Parameters of FEPSS in the experiments.**

| Parameters          | Descriptions                          | Values Used in |
|---------------------|---------------------------------------|-----------------|
| total time limit    | termination condition of Algorithm 2  | 360s, if \( n = 100 \) Algorithm 2 |
|                     |                                       | 3600s, if \( n \leq 250 \) Algorithm 2 |
|                     |                                       | 7200s, if \( n = 500 \) Algorithm 2 |
|                     |                                       | 36000s, if \( n > 500 \) Algorithm 2 |
| \( T \)             | time limit for each run of Integer Programming | 10s, if \( n \leq 500 \) Algorithm 7 |
|                     |                                       | 20s, if \( n > 500 \) Algorithm 7 |
| \( n_S \)           | number of solutions in the solution set \( SS \) | 100 Algorithm 3 |
| \( \maxIteration \) | number of iterations for the tabu search | 500 Algorithm 3 |
| \( \alpha \)        | parameter used in scoring function \( V(s) \) | 0.7 Algorithm 4 |
| \( \beta \)         | parameter used in scoring function \( \delta_i \) | 0.6 Algorithm 5 |
| \( P_1 \)           | probability used in randomFunction | 0.1 Algorithm 6 |
| \( P_2 \)           | probability used in ratioFunction | 0.7 Algorithm 6 |
| \( t_1 \)           | number of items added into the free part in explore procedure | 10 Algorithm 7 |
| \( t_2 \)           | number of items used in ratioFunction | 0.1 Algorithm 7 |
| \( \Delta \)        | parameter for the size of free part | 0.1 Algorithm 7 |
| \( \text{lnsLimit} \) | frequency of execution of LNS | 100 Algorithm 7 |
| \( \text{lnsIterationNum} \) | number of iterations in LNS | 10 Algorithm 7 |

**Table 4: Average results of OR-Library instances.**

| Instances | \( f_{\text{best}} \) | \( f_{\text{worst}} \) | \( f_{\text{avg}} \) | \( f_{\text{avg}}(\#) \) |
|-----------|-----------------|-----------------|-----------------|-----------------|
| \((100, 5)\) | 42640.4 | 42640.4 | 42640.4 | 42640.4 |
| \((100, 10)\) | 41606.0 | 41606.0 | 41606.0 | 41606.0 |
| \((250, 5)\) | 107088.9 | 107088.9 | 107088.9 | 107088.9 |
| \((250, 10)\) | 106365.7 | 106365.7 | 106365.7 | 106365.7 |
| \((500, 5)\) | 214168.8 | 214168.8 | 214168.8 | 214168.8 |
| \((500, 10)\) | 212859.3 | 212859.3 | 212859.3 | 212859.3 |
| \((500, 30)\) | 211453.7 | 211453.7 | 211453.7 | 211453.7 |
| Avg.       | 120185.3 | 210170.9 | 120176.9 | 120179.4 |
| #Best      | 4 | 4 | 4 | 4 |

from the Wilcoxon signed-rank tests indicate that the results of FEPSS significantly differ from those of other algorithms in terms of \( f_{\text{best}} \).

Finally, we present the results of MK, GK instances in Table 6. This set contains some large instances, e.g., \( n=2500 \) and \( m=100 \). We can see that our FEPSS finds new lower bounds for the largest 5 instances. It finds the best solutions for all the 11 instances. It also gets the best performance in the average objective of the 30 runs in 10 instances. Besides, it takes less time to find the best-found solutions in the largest instances than the others. From \( f_{\text{worst}} \), we can observe that our algorithm wins in 9 instances out of 11, along with the smallest average standard deviation indicating the stability of FEPSS.
| Instance | $I_{\text{best}}$ | $I_{\text{worst}}$ | $f_{\text{avg}}$ | $f_{\text{std}}$ | $t_{\text{avg}}(s)$ |
|----------|-----------------|-------------------|-----------------|-----------------|-----------------|
| 500-30-0 | 116056 | 116014 | 115952 | 115972 | 115924 |
| 500-30-1 | 114810 | 114810 | 114734 | 114769 | 114810 |
| 500-30-2 | 117641 | 117612 | 117612 | 116682 | 117641 |
| 500-30-3 | 115354 | 115357 | 115258 | 115313 | 115370 |
| 500-30-4 | 116525 | 116516 | 116437 | 116471 | 116539 |
| 500-30-5 | 115741 | 115741 | 115701 | 115734 | 115734 |
| 500-30-6 | 114181 | 114148 | 114085 | 114111 | 114181 |
| 500-30-7 | 114403 | 114314 | 114243 | 114248 | 114344 |
| 500-30-8 | 115419 | 115419 | 115419 | 115419 | 115419 |
| 500-30-9 | 117116 | 117116 | 117030 | 117104 | 117116 |
| 500-30-10 | 218104 | 218104 | 218053 | 218104 | 218104 |
| 500-30-11 | 214648 | 214629 | 214626 | 214645 | 214645 |
| 500-30-12 | 215978 | 215918 | 215905 | 215945 | 215945 |
| 500-30-13 | 217910 | 217892 | 217825 | 217862 | 217862 |
| 500-30-14 | 215689 | 215640 | 215649 | 215640 | 215640 |
| 500-30-15 | 215867 | 215852 | 215832 | 215852 | 215852 |
| 500-30-16 | 215907 | 215883 | 215897 | 215897 | 215897 |
| 500-30-17 | 216542 | 216510 | 216450 | 216542 | 216542 |
| 500-30-18 | 217340 | 217346 | 217329 | 217337 | 217364 |
| 500-30-19 | 214739 | 214739 | 214681 | 214701 | 214739 |
| 500-30-20 | 301675 | 301675 | 301643 | 301675 | 301675 |
| 500-30-21 | 300825 | 300825 | 300825 | 300825 | 300825 |
| 500-30-22 | 305087 | 305087 | 305087 | 305087 | 305087 |
| 500-30-23 | 302032 | 302032 | 302032 | 302032 | 302032 |
| 500-30-24 | 304462 | 304462 | 304462 | 304462 | 304462 |
| 500-30-25 | 297012 | 297012 | 296962 | 296998 | 297012 |
| 500-30-26 | 303364 | 303364 | 303364 | 303364 | 303364 |
| 500-30-27 | 307007 | 307007 | 307007 | 307007 | 307007 |
| 500-30-28 | 303199 | 303199 | 303199 | 303199 | 303199 |
| 500-30-29 | 300596 | 300596 | 300596 | 300596 | 300596 |

| avg. | 211451.9 | 211438.0 | 211399.2 | 211422.3 | 211448.0 |
|-------|----------|----------|----------|----------|----------|
| p-value | 1.02e-1 | 1.45e-3 | 5.23e-6 | 1.12e-3 |
| Instance  | (n, m) | \(f_{\text{best}}\) | \(f_{\text{worst}}\) | \(f_{\text{avg}}\) | Std. t | \(t_{\text{avg}(s)}\) |
|-----------|--------|------------------|------------------|------------------|--------|------------------|
| ma_gk01   | (100, 15) | 3766 3766 3766 3766 3766 3766 | 3766 3766 3766 3766 | 3766.0 3766.0 3766.0 | 48.4 4.8 | 131.5 96.6 |
| ma_gk02   | (100, 25) | 3958 3958 3958 3958 3958 3958 | 3958 3958 3958 3958 | 3958.0 3958.0 3958.0 | 7.9 4.8 | 2849.4 191.2 |
| ma_gk03   | (150, 25) | 5656 5656 5656 5656 5656 5656 | 5656 5656 5656 5656 | 5656.0 5656.0 5656.0 | 7.0 4.8 | 2849.4 191.2 |
| ma_gk04   | (150, 50) | 5767 5767 5767 5767 5767 5767 | 5767 5767 5767 5767 | 5767 5767 5767 | 0.7 1.5 | 3703.5 119.5 |
| ma_gk05   | (200, 25) | 7561 7561 7561 7561 7561 7561 | 7561 7561 7561 7561 | 7561 7561 7561 | 1.0 1.5 | 3703.5 119.5 |
| ma_gk06   | (200, 50) | 7680 7680 7680 7680 7680 7680 | 7680 7680 7680 7680 | 7680 7680 7680 | 1.5 1.7 | 3703.5 119.5 |
| ma_gk07   | (500, 25) | 19220 19220 19220 19220 19220 19220 | 19220 19220 19220 19220 | 19220 19220 19220 | 3.6 1.2 | 3703.5 119.5 |
| ma_gk08   | (500, 50) | 18806 18806 18806 18806 18806 18806 | 18806 18806 18806 18806 | 18806 18806 18806 | 1.4 1.2 | 3703.5 119.5 |
| ma_gk09   | (1500, 25) | 5507 5507 5507 5507 5507 5507 | 5507 5507 5507 5507 | 5507 5507 5507 | 2.4 2.0 | 3703.5 119.5 |
| ma_gk10   | (1500, 50) | 57295 57295 57295 57295 57295 57295 | 57295 57295 57295 57295 | 57301 57301 57301 | 9.1 2.0 | 2049.5 127.3 |
| ma_gk11   | (2500, 100) | 95237 95237 95237 95237 95237 95237 | 95237 95237 95237 95237 | 95237 95237 95237 | 2.1 1.5 | 7563.1 25669.3 |

Table 6: Detailed results of MK, GK instances.
6. Conclusion and future work

In this paper, we propose a novel algorithm that combines evolutionary computation with an exact algorithm for the 0-1 Multidimensional Knapsack Problem. The algorithm takes advantage of an efficient evolutionary algorithm to find promising search spaces, and then an exact algorithm is employed to explore the search spaces. We performed extensive experimentation with commonly used benchmark sets. The ablation study shows that the scoring function $\delta_i$ is effective in identifying the hard-to-decide items, so it is more efficient to find high-quality solutions in the promising search spaces. Consequently, our algorithm outperforms the state-of-the-art heuristics algorithms in solution quality. It finds a new lower bound for 10 hard instances. The new algorithm is more efficient in solving some hard and large 0-1 Multidimensional Knapsack Problems.

In the future, we would apply FEPS framework in solving other combinational optimization problems. In particular, we would focus on the problems with well-established exact algorithms and population-based evolutionary algorithms, as well as an objective function that can be calculated at a low cost. This is because the algorithm relies on a population to find good partial assignments that correspond to promising search space. If the population can not provide useful information, the algorithm will be less efficient.

7. Acknowledgements

This work is supported by National Natural Science Foundation of China under grant NO. 62276060, 61802056, 61976050 and Natural Science Foundation of Jilin Province under grant NO. 20210101470JC.

References

[1] B. Gavish, Allocation of databases and processors in a distributed computing system, Management of Distributed Data Processing (1982) 215–231.
[2] F. Chen, T. La Porta, M. B. Srivastava, Resource allocation with stochastic demands, in: 2012 IEEE 8th International Conference on Distributed Computing in Sensor Systems, IEEE, 2012, pp. 257–264.
[3] J. Zhang, N. Xie, X. Zhang, W. Li, An online auction mechanism for cloud computing resource allocation and pricing based on user evaluation and cost, Future Generation Computer Systems 89 (2018) 286–299.
[4] P. Gilmore, R. E. Gomory, The theory and computation of knapsack functions, Operations Research 14 (1966) 1045–1074.
[5] G. J. Beaujon, S. P. Marin, G. C. McDonald, Balancing and optimizing a portfolio of r&d projects, Naval Research Logistics (NRL) 48 (2001) 18–40.
[6] Discrete facility location and routing of obnoxious activities, Discrete Applied Mathematics 133 (2003) 3–28.
[7] J. Ajayi, A. Di Maio, T. Braun, D. Xenakis, An online multi-dimensional knapsack approach for slice admission control, in: 2023 IEEE 20th Consumer Communications & Networking Conference (CCNC), IEEE, 2023, pp. 152–157.
[8] A. P. Khandekar, A. Nargundkar, Dynamic programming approach to solve real-world application of multi-objective unbounded knapsack problem, in: Intelligent Systems and Applications: Select Proceedings of ICISA 2022, Springer, 2023, pp. 417–422.
[9] M. Harishankar, Incentivizing User-centric Resource Allocation in Wireless Networks in Realtime, Ph.D. thesis, Carnegie Mellon University, 2020.
[10] R. Mansini, M. G. Speranza, Coral: An exact algorithm for the multidimensional knapsack problem, INFORMS Journal on Computing 24 (2012) 399–415.
[11] F. Glover, G. A. Kochenberger, Critical event tabu search for multidimensional knapsack problems, in: Meta-heuristics, Springer, 1996, pp. 407–427.
[12] A. Drexl, A simulated annealing approach to the multiconstraint zero-one knapsack problem, Computing 40 (1988) 1–8.
[13] E. Angelelli, R. Mansini, M. G. Speranza, Kernel search: A general heuristic for the multi-dimensional knapsack problem, Computers & Operations Research 37 (2010) 2017–2026.
[14] F. Della Croce, A. Grosso, Improved core problem based heuristics for the 0/1 multi-dimensional knapsack problem, Computers & Operations Research 39 (2012) 27–31.
[15] M. Chih, C.-J. Lin, M.-S. Chern, T.-Y. Ou, Particle swarm optimization with time-varying acceleration coefficients for the multidimensional knapsack problem, Applied Mathematical Modelling 38 (2014) 1338–1350.

[16] M. Chih, Self-adaptive check and repair operator-based particle swarm optimization for the multidimensional knapsack problem, Applied Soft Computing 26 (2015) 378–389.

[17] M. Chih, Three pseudo-utility ratio-inspired particle swarm optimization with local search for multidimensional knapsack problem, Swarm and Evolutionary Computation 39 (2018) 279–296.

[18] B. Haddar, M. Khemakhem, S. Hanafi, C. Wilbaut, A hybrid quantum particle swarm optimization for the multidimensional knapsack problem, Engineering Applications of Artificial Intelligence 55 (2016) 1–13.

[19] X. Lai, J.-K. Hao, Z.-H. Fu, D. Yue, Diversity-preserving quantum particle swarm optimization for the multidimensional knapsack problem, Expert Systems with Applications 149 (2020) 113310.

[20] M. Chih, Stochastic stability analysis of particle swarm optimization with pseudo random number assignment strategy, European Journal of Operational Research 305 (2023) 562–593.

[21] S. Al-Shihabi, S. Ólafsson, A hybrid of nested partition, binary ant system, and linear programming for the multidimensional knapsack problem, Computers & Operations Research 37 (2010) 247–255.

[22] T. Nurcahyadi, C. Blum, Negative learning in ant colony optimization: Application to the multi dimensional knapsack problem, in: 2021 5th International Conference on Intelligent Systems, Metaheuristics & Swarm Intelligence, 2021, pp. 22–27.

[23] Y. Feng, G.-G. Wang, A binary moth search algorithm based on self-learning for multidimensional knapsack problems, Future Generation Computer Systems 126 (2022) 48–64.

[24] F. Setiawan, A. Sadiyoko, C. Setiardjo, Application of pigeon inspired optimization for multidimensional knapsack problem (2021).

[25] N. A. Al-Thanoon, O. S. Qasim, Z. Y. Algamal, A new hybrid pigeon-inspired optimization algorithm for solving multidimensional knapsack problems, in: 2021 7th International Conference on Contemporary Information Technology and Mathematics (ICCITM), IEEE, 2021, pp. 226–229.

[26] Y. Wei, An improved binary artificial bee colony algorithm for solving multidimensional knapsack problem, in: International Conference on Electronic Information Engineering and Computer Technology (EIECT 2021), volume 12087, SPIE, 2021, pp. 326–335.

[27] J. García, C. Maureira, A knn quantum cuckoo search algorithm applied to the multidimensional knapsack problem, Applied Soft Computing 102 (2021) 107077.

[28] T. Meng, Q.-K. Pan, An improved fruit fly optimization algorithm for solving the multidimensional knapsack problem, Applied Soft Computing 50 (2017) 79–93.

[29] L. Wang, X.-l. Zheng, S.-y. Wang, A novel binary fruit fly optimization algorithm for solving the multidimensional knapsack problem, Knowledge-Based Systems 48 (2013) 17–23.

[30] K. Luo, Q. Zhao, A binary grey wolf optimizer for the multidimensional knapsack problem, Applied Soft Computing 83 (2019) 105645.

[31] A. Rezoug, M. Bader-El-Den, D. Boughaci, Application of supervised machine learning methods on the multidimensional knapsack problem, Neural Processing Letters (2021) 1–20.

[32] J. García, E. Lalla-Ruiz, S. Voß, E. L. Droguett, Enhancing a machine learning binarization framework by perturbation operators: analysis on the multidimensional knapsack problem, International Journal of Machine Learning and Cybernetics 11 (2020) 1951–1970.

[33] J. García, P. Moraga, M. Valenzuela, H. Pinto, A db-scan hybrid algorithm: an application to the multidimensional knapsack problem, Mathematics 8 (2020) 507.

[34] Z. Li, L. Tang, J. Liu, A memetic algorithm based on probability learning for solving the multidimensional knapsack problem, IEEE Transactions on Cybernetics (2020).

[35] X. Lai, J.-K. Hao, F. Glover, Z. Lü, A two-phase tabu-evolutionary algorithm for the 0–1 multidimensional knapsack problem, Information sciences 436 (2018) 282–301.
[36] P. Shaw, Using constraint programming and local search methods to solve vehicle routing problems, in: Proc. CP’98, Springer, 1998, pp. 417–431.

[37] N. V. Nepomuceno, P. R. Pinheiro, A. L. Coelho, Combining metaheuristics and integer linear programming: a hybrid methodology applied to the container loading problem, in: Proceedings of the XX congreso da sociedade brasileira de computação, concurso de teses e dissertações, 2007, pp. 2028–32.

[38] N. Nepomuceno, P. Pinheiro, A. L. Coelho, Tackling the container loading problem: a hybrid approach based on integer linear programming and genetic algorithms, in: European Conference on Evolutionary Computation in Combinatorial Optimization, Springer, 2007, pp. 154–165.

[39] C. Blum, P. Pinacho, M. López-Ibáñez, J. A. Lozano, Construct, merge, solve & adapt a new general algorithm for combinatorial optimization, Computers & Operations Research 68 (2016) 75–88.

[40] J. E. Gallardo, C. Cotta, A. J. Fernández, Solving the multidimensional knapsack problem using an evolutionary algorithm hybridized with branch and bound, in: Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach: First International Work-Conference on the Interplay Between Natural and Artificial Computation, IWINAC 2005, Las Palmas, Canary Islands, Spain, June 15-18, 2005, Proceedings, Part II 1, Springer, 2005, pp. 21–30.

[41] H. Pirkul, A heuristic solution procedure for the multiconstraint zero-one knapsack problem, Naval Research Logistics (NRL) 34 (1987) 161–172.

[42] D. L. Woodruff, E. Zemel, Hashing vectors for tabu search, Annals of Operations Research 41 (1993) 123–137.

[43] T. Back, D. B. Fogel, Z. Michalewicz, Handbook of Evolutionary Computation, 1st ed., IOP Publishing Ltd., GBR, 1997.

[44] P. C. Chu, J. E. Beasley, A genetic algorithm for the multidimensional knapsack problem, Journal of heuristics 4 (1998) 63–86.

[45] M. Vasquez, J.-K. Hao, et al., A hybrid approach for the 0-1 multidimensional knapsack problem, in: IJCAI, Citeseer, 2001, pp. 328–333.

[46] M. Vasquez, Y. Vimont, Improved results on the 0–1 multidimensional knapsack problem, European Journal of Operational Research 165 (2005) 70–81.