mathematics

MDPI

Article

A Binary Machine Learning Cuckoo Search Algorithm Improved by a Local Search Operator for the Set-Union Knapsack Problem

José García [1,*], José Lemus-Romani [2], Francisco Altimiras [3,*], Broderick Crawford [4], Ricardo Soto [4], Marcelo Becerra-Rozas [4], Paola Moraga [1], Alex Paz Becerra [1], Alvaro Peña Fritz [1], Jose-Miguel Rubio [5] and Gino Astorga [6]

1 Escuela de Ingeniería en Construcción, Pontificia Universidad Católica de Valparaíso, Valparaíso 2362804, Chile; paola.moraga@pucv.cl (P.M.); alex.paz@pucv.cl (A.P.B.); alvaro.pena@pucv.cl (A.P.F.)
2 Escuela de Construcción Civil, Pontificia Universidad Católica de Chile, Santiago 7820436, Chile; jose.lemus@uc.cl
3 Facultad de Ingeniería y Negocios, Universidad de las Américas, Santiago 7500975, Chile
4 Escuela de Ingeniería Informática, Pontificia Universidad Católica de Valparaíso, Valparaíso 2362807, Chile; broderick.crawford@pucv.cl (B.C.); ricardo.soto@pucv.cl (R.S.); marcelo.becerra.r@mail.pucv.cl (M.B.-R.)
5 Facultad de Ingeniería, Ciencia y Tecnología, Universidad Bernardo O'Higgins Santiago, Metropolitana 8370993, Chile; josemiguel.rubio@ubo.cl
6 Escuela de Negocios Internacionales, Universidad de Valparaíso, Viña del Mar 2572048, Chile; gino.astorga@uv.cl
* Correspondence: jose.garcia@pucv.cl (J.G.); faltimiras@udla.cl (F.A.)

Abstract: Optimization techniques, specially metaheuristics, are constantly refined in order to decrease execution times, increase the quality of solutions, and address larger target cases. Hybridizing techniques are one of these strategies that are particularly noteworthy due to the breadth of applications. In this article, a hybrid algorithm is proposed that integrates the k-means algorithm to generate a binary version of the cuckoo search technique, and this is strengthened by a local search operator. The binary cuckoo search algorithm is applied to the $\mathcal{NP}$-hard Set-Union Knapsack Problem. This problem has recently attracted great attention from the operational research community due to the breadth of its applications and the difficulty it presents in solving medium and large instances. Numerical experiments were conducted to gain insight into the contribution of the final results of the k-means technique and the local search operator. Furthermore, a comparison to state-of-the-art algorithms is made. The results demonstrate that the hybrid algorithm consistently produces superior results in the majority of the analyzed medium instances, and its performance is competitive, but degrades in large instances.

Keywords: combinatorial optimization; machine learning; metaheuristics; set-union knapsack

Citation: García, J.; Lemus-Romani, J.; Altimiras, F.; Crawford, B.; Soto, R.; Becerra-Rozas, M.; Moraga, P.; Becerra, A.P.; Fritz, A.P.; Rubio, J.-M.; et al. A Binary Machine Learning Cuckoo Search Algorithm Improved by a Local Search Operator for the Set-Union Knapsack Problem. *Mathematics* **2021**, *9*, 2611. https://doi.org/10.3390/math9202611

Academic Editors: Anatoliy Swishchuk and Petr Stodola

Received: 9 September 2021
Accepted: 12 October 2021
Published: 16 October 2021

1. Introduction

Metaheuristics have demonstrated their efficacy in recent years in handling complex problems, especially complex combinatorial challenges. There are several examples in biology [1], logistics [2], civil engineering [3], and machine learning [4], among others. Despite the increased efficiency, and in part due to the vast scale of many combinatorial problems, it is also vital to maintain the strength of metaheuristic approaches. Thus, hybrid techniques have been employed to enhance metaheuristic algorithmic performance.

Among the main approaches of how to integrate metaheuristics, has been found hybrid heuristics, [5], where multiple metaheuristic algorithms are merged to boost their capabilities. In [6], for example, the authors employed simulated annealing-based genetic and tabu-search-based genetic algorithms to address the ordering planning problem. The

hybrid approaches were compared to the traditional approaches in this study, with the hybrid approaches outperforming the traditional approaches. In [7], the cuckoo search and firefly algorithm search methods are combined in order to avoid getting the procedure stuck in local optimum. The hybrid algorithm was applied to a job schedulers problem in high-performance computing systems. When compared to traditional policies, the results indicated significant reductions in server energy consumption.

Another interesting hybrid approach, [8], is matheuristics, which combines mathematical programming approaches with metaheuristic algorithms. The vehicle routing problem, for example, was studied utilizing mixed-integer linear programming and metaheuristic techniques in [9]. These methods, generally, do not take advantage of the auxiliary data created by metaheuristics in order to obtain more reliable results. In the solution-finding process, metaheuristics provide useful accessory data, which may be used to inform machine learning approaches. The area of artificial intelligence and in particular machine learning has grown important in recent times applying in different areas [10–12]. Machine learning approaches combined with metaheuristic algorithms is a novel area of research that has gained traction in recent years [13].

According to [13,14], there are three primary areas in which machine learning algorithms utilize metaheuristic data: low-level integrations, high-level integrations, and optimization problems. A current area of research in low-level integrations is the construction of binary versions of algorithms that operate naturally in continuous space. In [15], a state-of-the-art of the different binarization techniques is developed in which two main groups stand out. The first group corresponds to general binarization techniques in which the movements of the metaheuristics are not modified, but rather after its execution, the binarization of the solutions is applied. The second group corresponds to modifications applied directly to the movement of metaheuristics. The first group has the advantage that the procedure is used for any continuous metaheuristic, the second, when the adjustments are carried out in an adequate way, have good performance. There are examples of integration between machine learning and metaheuristics in this domain. In [16,17], the binary versions of the cuckoo search algorithm were generated using the k-nearest neighbor technique. These binary versions were applied to multidimensional knapsack and set covering problems, respectively. Whereas in the field of civil engineering [18,19], hybrid methods were proposed that utilizes db-scan and k-means, respectively, as a binarization method and is used to optimize the emission of CO_2 of retaining walls.

In accordance with low-level integration between machine learning and metaheuristics, in this article, a hybrid approach was used that combines a cuckoo search algorithm with the unsupervised k-means technique to obtain a binary version of the continuous cuckoo search algorithm. The suggested approach combines these two strategies with the objective of obtaining a robust binary version through the use of the data acquired during the execution of the metaheuristic. The proposed algorithm was applied to the set-union knapsack problem. The set union knapsack problem (SUKP) [20] is a generalization of the classical knapsack problem. SUKP has received attention from researchers in recent years [21–23] due to its interesting applications [24,25], as well as the difficulty of being able to solve it efficiently. In SUKP, there is a set of items where each item has a profit. Additionally, each item associates a set of elements where each element has a weight that is associated with the knapsack constraint. In the literature, it is observed that the algorithms that have addressed SUKP are mainly improved metaheuristics and have allowed obtaining results in reasonable times. When applying a metaheuristic in its standard form to SUKP, these algorithms have had limitations such as stability and decreased performance as the instance grows in size. For example, in [26], different transfer functions were used and evaluated with small and medium SUKP instances. This effect is observed when the algorithms are applied to standard SUKP instances, additionally, to increase the challenge in [27], a new set of benchmark problems was recently generated. All previous, leads to exploring hybrid techniques in order to strengthen the performance of the algorithm. The following are the contributions made by this work:

1. A new greedy initiation operator is proposed.
2. The k-means technique, proposed in [28], is used to binarize the cuckoo search (CS) algorithm, tuned and applied for the first time to the SUKP. Additionally, a random binarization operator is designed and two transition probabilities are applied to evaluate the contribution of k-means in the final result. It should be noted that the binarization method allows generating binary versions of other continuous swarm intelligence metaheuristics.
3. A new local search operator is proposed to improve the exploitation of the search space.
4. The results obtained by the hybrid algorithm are compared with different algorithms that have addressed SUKP. It should be noted that the standard SUKP instances and the new instances proposed in [27] were solved.

The following is a summary of the contents: Section 2 delves into the set-union knapsack problem and its applications. The k-means cuckoo search algorithm and the local search operator are described in Section 3. In Section 4, the detail of the numerical experiments and comparisons are developed. Finally, the conclusions and potential lines of research are discussed in Section 5.

2. The Set Union Knapsack Problem

The Set-Union Knapsack Problem (SUKP) is a generalized knapsack model with the following definition. First, let U be a set of n elements with each element $j \in U$ having a weight $w_j > 0$. Let V be a set of m items with each item $i \in V$ being a subset of elements $U_i \subseteq U$ and having a profit p_i. Finally, for a knapsack with capacity C, SUKP entails identifying a set of items $S \subseteq V$ that maximizes the total profit of S while guaranteeing that the total weight of the components of S does not exceed the capacity C of the knapsack. Being the elements belonging to the set S, the decision variables of the problem. It is worth noting that an elements weight is only tallied once, even if it corresponds to several chosen items in S. SUKP may be written mathematically as follows:

$$\text{Maximize } P(S) = \sum_{i \in S} p_i. \tag{1}$$

subject to:

$$W(S) = \sum_{j \in \cup_{i \in S} U_i} w_j \leq C, S \subseteq V. \tag{2}$$

In reviewing the literature SUKP has been found to have interesting applications, for example in [24]. The goal of this application is to improve the scalability of cybernetic systems robustness. Given a centralized cyber system with a fixed memory capacity that holds a collection of profit-generating services (or requests), each of which contains a set of data objects. When a data object is activated, it consumes a particular amount of memory, and using the same data object several times does not result in increased memory consumption (An important condition of SUKP). The goal is to choose a subset of services from among the candidate services that maximizes the total profit of those services while keeping the total memory required by the underlying data objects within the cyber system's memory capacity. The SUKP model, in which an item corresponds to a service with its profit and an element relates to a data object with its memory usage, is a convenient way to structure this application (element weight). Finding the optimal solution to the ensuing SUKP problem is thus comparable to solving the data allocation problem.

Another interesting application is related to the rendering of an animated crow in real-time [29]. In the article, the authors present a method to accelerate the visualization of large crowds of animated characters. They adopt a caching system that enables a skinned key-pose (elements) to be re-used by multi-pass rendering, between multiple agents and across multiple frames, an interpolative approach that enables key-pose blending to be supported. In this problem, each item corresponds to a crowd member. Applications are also found in data stream compression through the use of bloom filters [25].

SUKP is an $\mathcal{NP}$-hard problem [20] that has been tackled by a variety of methods. In [20,30], theoretical studies using greedy approaches or dynamic programming are found. An integer linear programming model was developed in [31] and applied to small instances of 85 and 100 items, finding the optimal solutions.

Metaheuristic algorithms have also addressed SUKP. In [32], the authors use an artificial bee colony technique to tackle SUKP. In addition, this algorithm integrates a greedy operator with the aim of addressing infeasible solutions. In [33], the authors designed an enhanced moth search algorithm. To improve its efficiency, this algorithm incorporates an integrating differential mutation operator. The Jaya algorithm was employed in [34]. Additionally, a differential evolution technique was incorporated to enhance exploration capability. The Cauchy mutation is used to boost its exploitation ability. Furthermore, an enhanced repair operator has been designed to repair the infeasible solutions. In [26], the effectiveness of different transfer functions is studied in order to binarize the moth metaheuristics. A local search operator is designed in [35] and applied to long-scale instances of SUKP. The article proposes three strategies that conform to the adaptive tabu search framework and efficiently solve new instances of SUKP. In [36], the grey wolf optimizer (GWO) algorithm is adapted to address binary problems. For the algorithm to be robust, traditional binarization methods are not used. To replicate the GWO leadership hierarchy technique, a multiple parent crossover is established with two distinct dominance tactics. In addition, an adaptive mutation with an exponentially decreasing step size is used to avoid early convergence and achieve a balance of intensification and diversification.

3. The Machine Learning Cuckoo Search Algorithm

This section describes the machine learning binary cuckoo search algorithm used to solve the SUKP problem. This hybrid algorithm consists of three main operators: A greedy initialization operator detailed in Section 3.1. CS is then used to develop the optimization. Here, it should be noted that CS is going to produce results with values in $\mathbb{R}$ and therefore they must be binarized. Then, a machine learning binarization operator performs the binarization of the solutions generated by the cuckoo search algorithm, and which uses the unsupervised k-means technique. This operator is detailed in Section 3.2. Finally, a local search operator is applied when the condition of finding a new maximum is met. The logic of the local search operator is detailed in Section 3.3. Figure 1 shows the flowchart of the binary machine learning cuckoo search algorithm. It is also worth noting that CS can be replaced by any other continuous swarm intelligence metaheuristic.

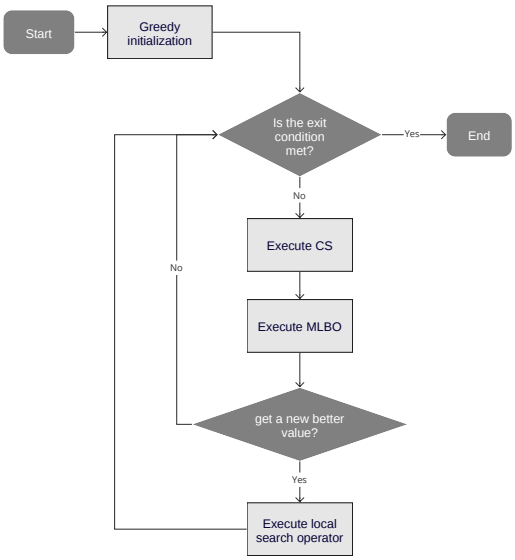

Figure 1. Machine learning cuckoo search binary algorithm.

3.1. Greedy Initialization Operator

The objective of this operator is to build the solutions that will start the search process. For this, the items are ordered using the ratio defined in Equation (3). As input to the operator, *sortItems* is utilized, and it contains the elements ordered by *r* from highest to lowest. As output, a valid solution, *Sol*, is obtained.

$$r = \frac{\text{item profit}}{\text{sum of element weights}} \tag{3}$$

In line 4, a blank *Sol* solution is initialized, then in line 5 the fulfillment of the constraint by *Sol* is validated. While the weight of the solution items (*weighSol*), Equation (2), is not equal or greater thab the knapsack constraint (*knapsackSize*), a random number *rand* is generated in line 6, and compare it in line 7 with β. If *rand* is greater than β, an element of *sortItems* is added in line 8, fulfilling the order. Otherwise, in line 11, a random item is chosen, then add it to the solution, and in line 12 remove it from *sortItems*. Once the knapsack is full, the solution needs to be cleaned up in line 15, as it is greater than or equal to *knapsackSize*. In the case that it is the same, it does not take action. In the event that it is greater, the items of *Sol* must be ordered using r defined in Equation (3) and it is removed in order starting with the smallest and checking the constraint in each elimination. Once the constraint is fulfilled, the procedure stops and the solution *Sol* is returned. The pseudo-code is shown in Algorithm 1.

Algorithm 1 Greedy initialization operator

```
1:  Function initSolutions(sortItems)
2:  Input sortItems
3:  Output Sol
4:  Sol ← []
5:  while (weightSol < knapsackSize) do
6:      rand ← getRandom()
7:      if rand > β then
8:          Sol ← addSortItem(sortItems)
9:          sortItems ← removeFromSortItems(Item)
10:     else
11:         Sol ← addRandomItem(sortItems)
12:         sortItems ← removeFromSortItems(Item)
13:     end if
14: end while
15: Sol ← cleanSol(Sol)
16: return Sol
```

3.2. Machine Learning Binarization Operator

The machine learning binarization operator (MLBO) is responsible for the binarization process. This receives as input the list *lSol* of solutions obtained from the previous iteration, the metaheuristic (*MH*), in this case CS, the best solution obtained, *bestSol* so far, and the transition probability for each cluster, *transProbs*. To this list *lSol*, in line 4, the *MH* is applied, in this case it corresponds to CS. From the result of applying *MH* to *lSol*, the absolute value of velocities, *vlSol*, is obtained. These velocities correspond to the transition vector obtained by applying MH to the list of solutions. The set of all velocities is clustered in line 5, using k-means (getKmeansClustering), in this particular case $K = 5$.

So, for each Sol_i and each dimension j, a cluster is assigned and each cluster is associated with a transition probability (*transProbs*), ordered by the value of the cluster centroid. For this case the transition probabilities used were [0.1, 0.2, 0.4, 0.8, 0.9]. Then for the set of points that belong to the cluster with the smallest centroid, which is represented

by the green color in Figure 2, the transition probability 0.1 was associated. For the group of blue points that obtained the centroid with the highest value, a transition probability of 0.9 was associated. The smaller the value of the centroid, the smaller the value of *transProbs* are associated with it. Then, in line 8, for each $lSol_{i,j}$, a transition probability $dimSolProb_{i,j}$ is associated and and later on line 9 compared with a random number r_1. In the case that $dimSolProb_{i,j} > r_1$, then it is updated considering the best value, line 10, and otherwise, it is not updated, line 12. Once all the solutions have been updated, each of them is cleaned up using the process explained in Section 3.1. In the case of a new best value is obtained, in line 19, a local search operator is executed. This local search operator is detailed in the following section. Finally, the updated list of solutions *lSol* and the best solution *bestSol* are returned. The pseudo-code is shown in Algorithm 2.

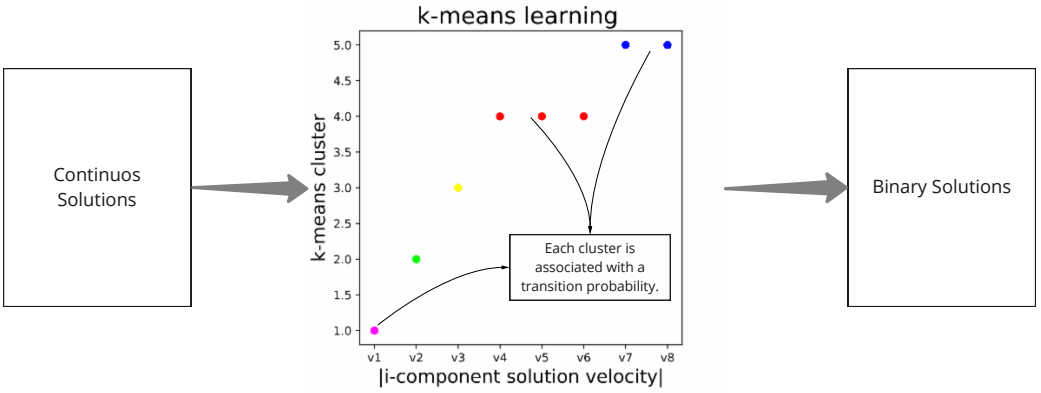

Figure 2. K-means binarization procedure.

Algorithm 2 Machine learning binarization operator (MLBO).

1: **Function** MLBO($lSol$, MH, $transProbs$, $bestSol$)
2: **Input** $lSol$, MH, $transProbs$
3: **Output** $lSol$, $bestSol$
4: $vlSol \leftarrow$ getAbsValueVelocities($lSol$, MH)
5: $lSolClust \leftarrow$ getKmeansClustering($vlSol$, K)
6: **for** (each Sol_i in $lSolClust$) **do**
7: **for** (each $dimSol_{i,j}l$ in Sol_i) **do**
8: $dimSolProb_{i,j}$ = getClusterProbability($dimSol$, $transProbs$)
9: **if** $dimSolProb_{i,j} > r_1$ **then**
10: Update $lSol_{i,j}$ considering the best.
11: **else**
12: Do not update the item in $lSol_{i,j}$
13: **end if**
14: **end for**
15: $Sol_i \leftarrow$ cleanSol(Sol_i)
16: **end for**
17: $tempBest \leftarrow$ getBest($lSol$)
18: **if** $cost(tempBest) > cost(bestSol)$ **then**
19: $tempBest \leftarrow$ execLocalSearch($tempBest$)
20: $bestSol \leftarrow tempBest$
21: **end if**
22: **return** $lSol$, $bestSol$

3.3. Local Search Operator

According to Figure 1, the local search operator is executed every time the metaheuristic finds a new best value. As input, the local search operator receives the new best values (*bestSol*), and as a first stage, it uses it to obtain the items that belong and do not belong to *bestSol*, line 4 of Algorithm 3. These two lists of items are iterated, $T = 300$ times, performing a swap without repetition, line 7 of Algorithm 3. Once the swap is carried out, the conditions are evaluated: it will improve the profit and that the weight of the knapsack is less than or equal to *knapsackSize*. If both conditions are met, the *bestSol* is updated by *tempSol*, to finally return *bestSol*.

Algorithm 3 Local search.

```
1:  Function LocalSearch(bestSol)
2:  Input bestSol
3:  Output bestSol
4:  lsolItems, lsolNoItems ← getItems(bestSol)
5:  i = 0
6:  while (i < T) do
7:      tempSol ← swap(lsolItems, lsolNoItems)
8:      if profit(tempSol) > profit(bestSol) and knapsack(tempSol) <= knapsackSize
        then
9:          bestSol ← tempSol
10:     end if
11:     i += 1
12: end while
13: return bestSol
```

4. Results

This section details the experiments conducted with MLBO and cuckoo search metaheuristic, to determine the proposed algorithms effectiveness and contribution when applied to a $\mathcal{NP}$-hard combinatorial problem. This specific version of MLBO that cuckoo search uses will be denoted by MLCSBO. The SUKP was chosen as a benchmark problem because it has been approached by several algorithms and is not trivial to solve in small, medium and large instances. However, it should be emphasized that the MLBO binarization technique is easily adaptable to other optimization algorithms. The optimization algorithm chosen was CS because it is a simple-to-parameterize algorithm that has been used to solve a wide variety of optimization problems.

Python 3.6 was used to build the algorithm, as well as a PC running Windows 10 with a Core i7 processor and 16 GB of RAM. To evaluate whether the difference is statistically significant, the Wilcoxon signed-rank test was used. Additionally, 0.05 was utilized as the significance level. The test is chosen in accordance with the methodology outlined in [37,38]. The Shapiro–Wilk normality test is used initially in this process. If one of the populations is not normal and both have the same number of points, the Wilcoxon signed-rank test is proposed to determine the difference. In the experiments, the Wilcoxon test was used to compare the MLCSBO results with the other variants or algorithms used in pairs. For comparison, the complete list of results was always used. Further, in the case of the experiment in Section 4.2, since there are multiple comparisons and in order to correct for these comparisons, a post hoc test was performed with the Holm–Bonferroni correction. The statsmodels and scipy libraries of Python were used to develop the tests. Each instance was resolved 30 times in order to acquire the best value and average indicators. Additionally, the average time (in seconds) required for the algorithm to find the optimal solution is reported for each instance.

The first set of instances were proposed in [39]. These instances have between 85 and 500 items and elements. These instances are characterized by two parameters. A first parameter $\mu = (\sum_{i=1}^{m} \sum_{j=1}^{n} R_{ij})/(mn)$, which represents the density in the matrix, where $R_{ij} = 1$ means the item i includes to the j element. A second parameter $\nu = C/(\sum_{j=1}^{n} w_j)$, which represents the capacity ratio C over the total weight of the elements. Then, a SUKP instance is named as $m\_n\_\mu\_\nu$. The second group of instances was introduced in [27], and in this case, they contain between 585 and 1000 items and elements. The form was built following the same previous structure.

4.1. Parameter Setting

The methods described in [28,40] was used to pick the parameters. To make an appropriate parameter selection, this methodology employs four metrics specified by the Equations (4)–(7). Values were generated using the instances 100_85_0.10_0.75, 100_100_0.15_0.85, and 85_100_0.10_0.75. Each parameter combination was run ten times. The collection of parameters that have been explored and selected is presented in Table 1. To determine the configuration, the polygon area obtained from the four metric radar chart is calculated for each setting. The configuration that obtained the largest area was selected. In the case of the transition probabilities, only the probability of the third cluster was varied considering the values [0.4, 0.5], the rest of the values were considered constant.

1. The difference in percentage terms between the best value achieved and the best known value:
$$bSolution = 1 - \frac{KnownBestValue - BestValue}{KnownBestValue} \quad (4)$$
2. The percentage difference between the worst value achieved and the best value known:
$$wSol = 1 - \frac{KnownBestValue - WorstValue}{KnownBestValue} \quad (5)$$
3. The percentage departure of the obtained average value from the best-known value:
$$aSol = 1 - \frac{KnownBestValue - AverageValue}{KnownBestValue} \quad (6)$$
4. The convergence time used in the execution:
$$nTime = 1 - \frac{convergenceTime - minTime}{maxTime - minTime} \quad (7)$$

Table 1. Parameter setting for the MLCSBO.

Parameters	Description	Value	Range
N	Number of Nest	20	[10, 15, 20]
K	Clusters number	5	[4, 5, 6]
γ	Step Length	0.01	0.01
κ	Levy distribution parameter	1.5	1.5
T	Maximum local search iterations	300	[300, 400, 800]
β	Random initialization parameter	0.3	[0.3, 0.5]
Transition probability	Transition probability	[0.1, 0.2, 0.4, 0.8, 0.9]	[0.1, 0.2, [0.4, 0.5], 0.8, 0.9]

4.2. Insight into Binary Algorithm

The objective of this section is to determine the contribution of the MLCSBO operator and the local search operator in the final result of the optimization. To address this challenge, a random operator is designed that aims to replace MLBO in Figure 1 with an operator that performs random transitions. In particular, two configurations are studied Random-05, which has a 50% chance of making a transition, and Random-03, which has

a 30% chance of making a transition. Additionally, the configuration with and without a local search operator is studied. Each of the algorithms is evaluated for its performance without (NL) and with the local search operator.

The results are shown in Tables 2 and 3 and Figure 3. From Table 2, it can be deduced that the best values obtained are for MLCSBO, which has the binarization mechanism based on k-means. The above for both indicators average and best value. When comparing MLCSBO-NL, note that MLCSBO-NL does not have the local search operator, with Random-03-NL and Random-05-NL, it is noted that MLCSBO-NL is more robust in the averages and best values. This allows evaluating the effect of incorporating k-means with respect to a random binarization operator in the optimization result. Furthermore, MLCSBO-NL works better than Random-03 and Random-05, where the latter incorporate the local search operator. On the other hand, when analyzing the contribution of the local operator, it is observed that each time it is incorporated generates an improvement in both the averages and the best values. First the Wilcoxon statistical test was applied, where MLCSBO is compared with the other variations. The statistical test indicates that the differences are significant between MLCSBO and the other variations analyzed. However, as there are multiple comparisons, the *p*-values were corrected using the Holm–Bonferroni test. For this correction, the experiments of the operators Random-03 and Random-05 were treated as independent groups. In Figure 3, we see that the highest time is for MLCSBO. In particular, Random-05-NL, which corresponds on average to the best performer, is 18.8% faster than MLCSBO. On the other hand, MLCSBO-NL which does not have the local search operator is 7.4% faster than MLCSBO.

In Figure 3 and Table 4, the *%-Gap*, defined in Equation (8), with respect to the best known value is compared of the different variants developed in this experiment. The comparison is made through box plots. In Figure, it is observed that MLCSBO has a more robust behavior than the rest, since it obtains better values and smaller dispersions than the other variants. On the other hand, the variants that obtain the worst performance correspond to those that have the random binarization operator and do not use the local search operator.

$$\%\text{-}Gap = 100 \times \frac{BestknownValue - Value}{BestknownValue} \tag{8}$$

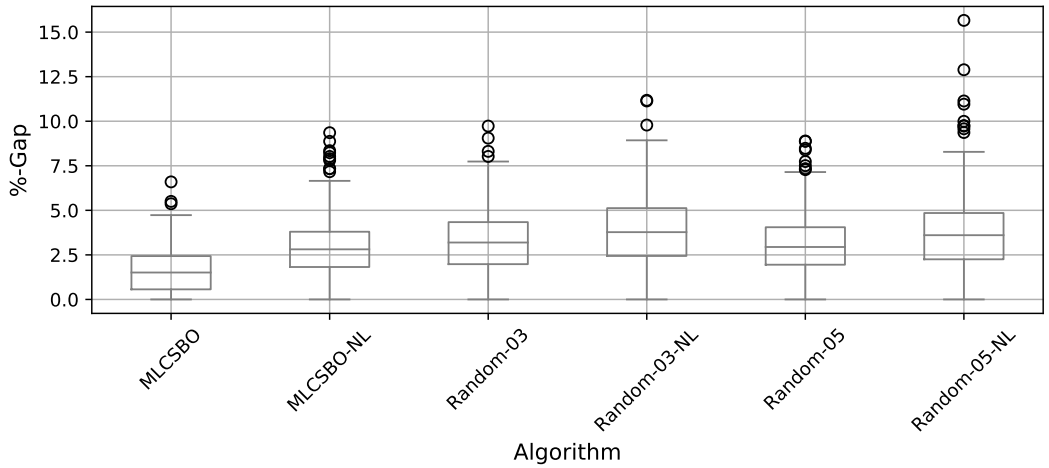

Figure 3. Box plots for MLCSBO and random operators, with and without local search operator.

Table 2. Comparison between MLCSBO and random operators, with and without local search operator.

	Random-03			Random-05			Random-03-NL			Random-05-NL			MLCSBO-NL			MLCSBO		
Instance	**Avg**	**Best**	**Std**	**Avg**	**Best**	**Std**	**Avg**	**Best**	**Std**	**Avg**	**Best**	**Std**	**Avg**	**Best**	**Std**	**Avg**	**Best**	**Std**
100_85_0.10_0.75	12,825	13,089	200.9	12,862.5	13,089	136.3	12,779.4	13,044	243.8	12,794.4	13,089	207	12,927.8	13,089	93	13,060	13,283	46.4
100_85_0.15_0.85	12,071.6	12,233	83.7	12,128	12,233	58.3	12,063.7	12,226	100.6	12,039.4	12,272	106.8	12,177	12,274	46.3	12,237.6	12,274	18.2
200_185_0.10_0.75	13,296.4	13,502	89.4	13,296.5	13,443	102	13,265	13,521	112.7	13,252.1	13,405	94.7	13,357.1	13,521	72.8	13,429.8	13,521	44.1
200_185_0.15_0.85	13,701.3	14,215	238.2	13,711.2	13,995	163.4	13,624	14,102	226.9	13,581.9	13,979	181	13,729.5	14,187	165.2	13,853.8	14,215	149.9
300_285_0.10_0.75	11,221.8	11,469	118.8	11,282.7	11,563	104.9	11,219.3	11,545	167.4	11,218.9	11,545	165.8	11,305.7	11,563	110.1	11,419.4	11,563	70.8
300_285_0.15_0.85	12,082.2	12,402	158.4	12,114.3	12,380	121.7	11,942.9	12,273	190.6	11,960.3	12,402	221.6	12,116.1	12,380	119	12,263.4	12,402	61.7
400_385_0.10_0.75	11,282	11,484	99.9	11,333.2	11,484	84.4	11,241.5	11,484	109.2	11,273.4	11,484	117.8	11,295.1	11,484	83	11,461.3	11,484	48.9
400_385_0.15_0.85	10,716.3	11,209	209.7	10,836.5	11,209	180.9	10,677.7	11,209	238	10,598.7	10,923	188.8	10,837.5	11,209	179.1	10,971.8	11,209	164.4
500_485_0.10_0.75	11,467.5	11,658	101.3	11,530	11,689	96.8	11,416.7	11,610	109.1	11,507.2	11,729	128	11,554.1	11,722	74	11,636.2	11,729	38.3
500_485_0.15_0.85	9779.8	10,217	126.2	9783.5	10,217	136.1	9695.4	10,217	144.9	9686.4	10,086	154.9	9811.4	10,086	98.1	9916.4	10,217	99.8
100_100_0.10_0.75	13,831.7	13,957	105.2	13,835.2	13,957	74.8	13,698.6	13,957	165.6	13,686.1	13,937	160	13,856.8	13,957	70.2	13,952.8	13,990	11.4
100_100_0.15_0.85	13,133.5	13,445	182.4	13,157	13,498	180.5	13,057.2	13,407	181.2	13,013.1	13,449	249.9	13,180.4	13,498	173.1	13,337.8	13,508	148.3
200_200_0.10_0.75	12,180.8	12,350	87.2	12,181.3	12,522	106.1	12,136.7	12,384	126.1	12,088.6	12,301	124.7	12,196.1	12,522	110.8	12,330.6	12,522	101.1
200_200_0.15_0.85	11,793	12,317	153.1	11,797.1	12,048	165.1	11,663.7	11,930	156.3	11,681.4	12,100	194.6	11,757.8	11,982	118.7	11,975.1	12,317	140.4
300_300_0.10_0.75	12,539.5	12,817	90.9	12,578.8	12,817	98.3	12,526	12,736	114.9	12,465.1	12,817	162.1	12,621.5	12,817	85.3	12,716.4	12,817	69.2
300_300_0.15_0.85	11,157.8	11,425	138.8	11,240.7	11,410	98.9	11,137.4	11,410	200	11,042.9	11,410	178.1	11,231.6	11,410	97.8	11,408.8	11,425	26.7
400_400_0.10_0.75	11,397.7	11,665	154.8	11,406.7	11,665	134.3	11,328.7	11,665	147.5	11,378.2	11,665	143.5	11,415.1	11,665	102.8	11,600.9	11,665	73.9
400_400_0.15_0.85	11,046.4	11,325	166.9	11,087.2	11,325	156.6	10,911.2	11,325	244.5	10,936.8	11,325	258.7	11,090.9	11,325	122.8	11,271.2	11,325	62.4
500_500_0.10_0.75	10,753.2	10,943	86.1	10,841.6	11,041	85.5	10,748.9	11,078	148.5	10,769.5	11,011	121.2	10,846.4	10,983	70	10,954.3	11,078	74.2
500_500_0.15_0.85	9847.3	10,108	146.1	9874.6	10,194	163.3	9768.1	10,160	191.4	9829.9	10,209	188	9879.4	10,162	146.5	10,056.6	10,209	107.2
85_100_0.10_0.75	11,761.2	12,045	159.6	11,797.4	12,045	131.7	11,684.1	11,964	181.9	11,713.4	12,045	160	11,797.7	12,045	159.8	11,945.5	12,045	126.6
85_100_0.15_0.85	11,994.5	12,369	223.4	12,053	12,299	129.2	11,942.5	12,348	235.5	11,988	12,369	233.6	12,083	12,369	152.9	12,253.1	12,369	81.6
185_200_0.10_0.75	13,539.9	13,659	99.6	13,493.7	13,695	90.8	13,468	13,696	120	13,449.8	13,695	125.1	13,558.7	13,696	78.7	13,651.7	13,696	36.1
185_200_0.15_0.85	10,890.6	11,155	94.4	10,883.2	11,298	164.6	10,893.9	11,242	135	10,873.8	11,298	149.9	10,969.7	11,298	139.4	11,068.7	11,298	162.4
285_300_0.10_0.75	11,401.4	11,568	100.4	11,407.4	11,568	105.4	11,323.9	11,568	146.6	11,359.6	11,568	126.7	11,410.1	11,568	83.3	11,546	11,568	13.5
285_300_0.15_0.85	11,258.4	11,763	221.4	11,368.1	11,763	149.9	11,220.4	11,714	225.8	11,193.4	11,590	240.1	11,333.5	11,763	208.4	11,564.3	11,763	129.2
385_400_0.10_0.75	10,274.1	10,397	69.5	10,300.6	10,407	71.2	10,248.8	10,436	100.3	10,249.1	10,467	88	10,302	10,407	63.7	10,400.7	10,600	47.3
385_400_0.15_0.85	9918.2	10,506	242.8	9921.8	10,294	212.1	9815	10,354	253.2	9771	10,329	358.2	9955.1	10,506	274.9	10,162	10,506	172.7
485_500_0.10_0.75	10,823.3	11,094	83.1	10,828.5	11,115	100.5	10,790.6	11,097	135	10,785.8	11,097	97.6	10,895.6	11,115	105.4	10,965.3	11,125	96.2
485_500_0.15_0.85	9836.6	10,117	162.2	9873.8	10,208	150.7	9760.3	10,104	160.1	9795.1	10,220	191.9	9897.4	10,104	122	10,095.7	10,220	74.1
Average	11,594.1	11,883.4	139.8	11,626.9	11,882	125.1	11,535.0	11,860	167.1	11,532.8	11,860	170.6	11,646.3	11,890	117.6	11,783.6	11,931	83.2
p-value Wilcoxon	1.7×10^{-6}	6.5×10^{-4}		1.8×10^{-6}	1.9×10^{-4}		1.7×10^{-6}	5.9×10^{-5}		1.7×10^{-6}	1.9×10^{-4}		1.7×10^{-6}	9.7×10^{-4}				
p-value Holm–Bonferroni	5.1×10^{-6}	0.0013		5.4×10^{-6}	3.8×10^{-4}		5.1×10^{-6}	1.2×10^{-4}		5.1×10^{-6}	3.8×10^{-4}		1.7×10^{-6}	9.7×10^{-4}				

Table 3. Average runtime values in seconds for MLCSBO and random operators, with and without a local search operator.

Instance	MLCSBO	MLCSBO-NL	Random-03	Random-03-NL	Random-05	Random-05-NL
100_85_0.10_0.75	10	9	10	7	10	5
100_85_0.15_0.85	20	18	19	16	18	15
200_185_0.10_0.75	26	21	17	15	19	14
200_185_0.15_0.85	61	48	54	40	54	51
300_285_0.10_0.75	33	25	22	15	23	17
300_285_0.15_0.85	66	60	55	57	69	48
400_385_0.10_0.75	35	32	31	26	32	29
400_385_0.15_0.85	97	102	106	101	101	99
500_485_0.10_0.75	63	55	39	43	56	47
500_485_0.15_0.85	119	106	107	94	111	112
100_100_0.10_0.75	6	5	4	4	4	5
100_100_0.15_0.85	17	17	12	14	16	14
200_200_0.10_0.75	32	31	24	25	24	24
200_200_0.15_0.85	140	143	124	122	133	112
300_300_0.10_0.75	99	95	88	84	87	85
300_300_0.15_0.85	156	148	160	152	153	144
400_400_0.10_0.75	46	43	32	30	42	37
400_400_0.15_0.85	202	198	176	161	191	143
500_500_0.10_0.75	82	89	80	69	84	80
500_500_0.15_0.85	168	135	153	128	123	110
85_100_0.10_0.75	4	4	4	4	4	4
85_100_0.15_0.85	18	17	18	17	18	18
185_200_0.10_0.75	29	32	29	25	29	20
185_200_0.15_0.85	67	53	42	43	47	44
285_300_0.10_0.75	20	18	16	17	14	16
285_300_0.15_0.85	91	79	73	66	67	60
385_400_0.10_0.75	72	65	53	70	57	69
385_400_0.15_0.85	122	101	92	92	85	99
485_500_0.10_0.75	114	114	108	104	87	108
485_500_0.15_0.85	130	123	120	106	134	110
Average	71.5	66.2	62.3	58.2	63.1	58.0

Table 4. Percentile values for MLCSBO and Random operators, with and without local search operator.

Percentile	MLCSBO	MLCSBO-NL	Random-03	Random-03-NL	Random-05	Random-05-NL
2.5	0.00	0.01	0.16	0.63	0.02	0.22
25	0.57	1.82	1.98	2.44	1.95	2.25
50	1.51	2.81	3.19	3.78	2.94	3.60
75	2.43	3.80	4.34	5.12	4.05	4.85
97.5	4.06	6.08	6.64	7.64	6.25	7.61

Additionally, the significance has been analyzed using the Wilcoxon test for the other variants. The details of the results are shown in Table 5. In each cell of the table, the *p*-values of best | average are written. In the table, it is observed that the difference of MLCSBO-NL with respect to the Random variants is not significant in the best indicator, but it is significant in the average indicator. The same goes for Random03 with respect to Random05. However, when analyzing Random03-NL with respect to Random05-NL, there is no significant difference in any of the indicators.

Table 5. Best | average *p*-values for the Wilcoxon test.

	MLCSBO	MLCSBO-NL	Random-03	Random-03-NL	Random-05
MLCSBO	-				
MLCSBO-NL	9.7×10^{-4} | 1.7×10^{-6}	-			
Random-03	6.5×10^{-4} | 5.1×10^{-6}	0.31 | 5.7×10^{-5}	-		
Random-03-NL	1.2×10^{-4} | 5.1×10^{-6}	0.06 | 1.7×10^{-5}	0.23 | 2.1×10^{-5}	-	
Random-05	3.8×10^{-4} | 5.4×10^{-6}	0.97 | 0.0012	0.61 | 3.1×10^{-4}	0.05 | 1.9×10^{-5}	-
Random-05-NL	3.8×10^{-4} | 5.1×10^{-6}	0.44 | 1.7×10^{-5}	0.47 | 8.4×10^{-4}	0.77 | 0.87	0.50 | 1.7×10^{-5}

4.3. Algorithm Comparisons

This section compares MLCSBO performance to that of other algorithms that have tackled SUKP. Different forms of approximations were used in the comparative selection. A genetic algorithm (GA), in which uniform mutation, point cross-over, and roulette wheel selection operators were used. In particular, the cross-over probability was $p_c = 0.8$ and the mutation probability was selected at $p_m = 0.01$. An artificial bee colony (ABC_{bin}, BABC), where the parameters used were a = 5 and limit defined as $Max\{m, n\}/5$, and a binary evolution technique (binDE) with factor $F = 0.5$ and crossover constant in 0.3, were adapted in [32] to tackle the SUKP. In [41], a weighted superposition attraction algorithm (bSWA), with parameters $\tau = 0.8$, $\phi = 0.008$, and $sl^1 = 0.4$, is proposed to solve SUKP. Two variations gPSO and gPSO* of particle swam optimization algorithm were proposed in [22]. In the case of gPSO, the init parameters used were $r^1 = 0.05$, $\phi = 0.005$, $p_1 = 0.2$, and $p_2 = 0.8$. In the case of gPSO*, $p_1 = 0.10$ and $p_2 = 0.70$. An artificial search agent with cognitive intelligence (intAgents) was proposed in [42], where the parameters used are, $\theta_{mut} = 0.005$, $m_{rate} = 0.05$, $p_{xover} = 0.6$, and $p_{xover}^{itMax} = 0.1$. Finally, the DH-Jaya algorithm was designed in [34], with parameters $C_r = 0.8$, $F + 0.8$, and $C_a = 1$. In Tables 6 and 7, the comparisons of the 30 smallest instances of SUKP are presented. Table 8 shows the results for the 30 largest instances. In the latter case, only results were found for BABC and DH-Jaya reported in the literature.

Consider Tables 6 and 7, which summarize the results for the 30 smallest instances. MLCSBO had the best value in 27 of the 30 cases. After that, GWOrbd has 16 best values, DH-Jaya has 11 best values, and GWOfbd also has 11 best values. It is possible that more than one algorithm gets the best value in some cases, in which case they are repeated in the accounting. This shows a good performance of the MLCSBO algorithm with respect to the other algorithms both in finding the best values as well as in reproducing these systematically. However, when the results for the 30 largest instances are analyzed, which are shown in Table 8, it is observed that the good performance obtained by MLCSBO is not repeated. In the case of larger instances, DH-Jaya is observed to perform better than MLCSBO. In the case of the best value indicator, DH-Jaya obtains 22 best values and MLCSBO 10. Even more, so when the average indicator is compared, MLCSBO gets 4 best averages and DH-Jaya, 26. To make sure there was an exploit problem on the local search operator, in these cases T was changed to 800, however, no improvements were obtained. The latter raises the suspicion that the decrease in performance in large cases is related to the exploration of the algorithm.

Table 6. Comparison between GA, BABC, ABC bin, gPSO*, gPSO, intAgents, DH-Jaya, GWOfbd, GWOrbd and MLCSBO algorithms for medium instances.

Instance	Results	Best Known	GA	BABC	ABC_{bin}	binDE	bWSA	gPSO*	gPSO	intAgents	DH-jaya	GWOfbd	GWOrbd	MLCSBO
100_85_0.10_0.75	best	13,283	13,044	13,251	13,044	13,044	13,044	13,167	**13,283**	**13,283**	**13,283**	13,089	**13,283**	**13,283**
	Avg		12,956.4	13,028.5	12,818.5	12,991	12,915.67	12,937.05	13,050.53	13,061.02	13,076	**13,041.37**	13,065.93	13,060
	std dev		130.66	92.63	153.06	75.95	185.45	189.6	37.41	44.08	66.61	31.17	70.02	46.4
100_85_0.15_0.85	best	12,479	12,066	12,238	12,238	**12,274**	12,238	12,210	**12,274**	**12,274**	**12,274**	**12,274**	**12,274**	**12,274**
	Avg		11,546	12,155	12,049.3	12,123.9	11,527.41	11,777.71	12,084.82	12,074.84	12,192.5	12,079.4	12,053.57	**12,237.6**
	std dev		214.94	53.29	96.11	67.61	332.27	277.16	95.38	86.37	70.25	93.34	81.99	18.2
200_185_0.10_0.75	best	13,521	13,064	13,241	12,946	13,241	13,250	13,302	13,405	13,502	13,405	13,405	13,405	**13,521**
	Avg		12,492.5	13,064.4	11,861.5	12,940.7	12,657.65	12,766.38	13,286.56	13,226.28	13,306.6	13,282.3	13,280.78	**13,429.8**
	std dev		320.03	99.57	324.65	205.7	319.58	304.82	93.18	150.92	60.96	102.88	123.63	44.1
200_185_0.15_0.85	best	14,215	13,671	13,829	13,671	13,671	13,858	13,993	14,044	14,044	**14,215**	**14,215**	**14,215**	**14,215**
	Avg		12,802.9	13,359.2	12,537	13,110	12,585.35	12,949.05	13,492.6	13,441.06	13,660.2	13,464.35	13,479.99	**13,853.8**
	std dev		291.66	234.99	289.53	269.69	302.66	325.58	328.72	324.96	274.76	358.97	358.56	149.9
300_285_0.10_0.75	best	11,563	10,553	10,428	9751	10,420	10,991	10,600	11,335	11,335	10,934	11,413	11,335	**11,563**
	Avg		9980.87	9994.76	9339.3	9899.24	10,366.21	10,090.47	10,669.51	10,576.1	10,703.2	10,707.54	10,684.17	**11,419.4**
	std dev		142.97	154.03	158.15	153.18	257.1	236.14	227.85	281.13	112.95	230.46	242.43	70.8
300_285_0.15_0.85	best	12,607	11,016	12,012	10,913	11,661	12,093	11,935	12,245	12,247	12,245	**12,402**	12,259	**12,402**
	Avg		10,349.8	10,902.9	9957.85	10,499.4	10,901.59	10,750.3	11,607.1	11,490.26	12,037.5	11,646.23	11,606.32	**12,263.4**
	std dev		215.13	449.45	276.9	403.95	508.79	524.53	477.8	518.81	296.02	517.63	492.99	61.7
400_385_0.10_0.75	best	11,484	10,083	10,766	9674	10,576	11,321	10,698	**11,484**	**11,484**	11,337	**11,484**	**11,484**	**11,484**
	Avg		9641.85	10,065.2	9187.76	9681.46	10,785.74	9946.96	10,915.87	10,734.62	11,062	10,884.49	10,880.22	**11,461.3**
	std dev		168.94	241.45	167.08	275.05	361.45	295.28	367.75	371.37	273.63	396.92	386.79	48.9
400_385_0.15_0.85	best	11,209	9831	9649	8978	9649	10,435	10,168	10,710	10,710	10,431	10,710	10,757	**11,209**
	Avg		9326.77	9135.98	8539.95	9020.87	9587.72	9417.2	9864.55	9735	10,017.9	9894.54	9900.01	**10,971.8**
	std dev		192.2	151.9	161.83	150.99	360.29	360.03	315.38	370.44	207.98	329.34	325.03	164.4
500_485_0.10_0.75	best	11,771	11,031	10,784	10,340	10,586	11,540	11,258	11,722	11,722	11,722	11,722	**11,771**	11,729
	Avg		10,567.9	10,452.2	9910.32	10,363.8	10,921.58	10,565.9	11,184.51	11,111.63	11,269.4	11,276.49	11,338.26	**11,636.2**
	std dev		123.15	114.35	120.82	93.39	351.69	260.32	322.98	355.18	275.37	347.99	351.46	38.3
500_485_0.15_0.85	best	10,238	9472	9090	8759	9191	9681	9756	10,022	10,059	9770	10,194	10,194	**10,217**
	Avg		8692.67	8857.89	8365.04	8783.99	9013.09	8779.44	9299.56	9165.26	9354.28	9339.8	9398.07	**9916.4**
	std dev		180.12	94.55	114.1	131.05	204.85	300.11	277.62	282.55	212.69	252.98	266.46	99.8
100_100_0.10_0.75	best	14,044	**14,044**	13,860	13,860	13,814	**14,044**	13,963	**14,044**	**14,044**	**14,044**	**14,044**	**14,044**	13,990
	Avg		13,806	13,734.9	13,547.2	13,675.9	13,492.71	13,739.71	13,854.71	13,767.23	13,912.5	13,861.35	13,847.86	**13,952.8**
	std dev		144.91	70.76	119.11	119.53	325.34	119.52	96.23	131.59	84.55	84.62	100.33	11.4
100_100_0.15_0.85	best	13,508	13,145	**13,508**	13,498	13,407	13,407	13,498	**13,508**	**13,508**	**13,508**	**13,508**	**13,508**	**13,508**
	Avg		12,234.8	13,352.4	13,103.1	13,212.8	12,487.88	12,937.53	13,347.58	13,003.62	**13,439.1**	13,312.57	13,297.37	13,337.8
	std dev		388.66	155.14	343.46	287.45	718.23	417.91	194.34	375.74	44.86	189.12	172.16	148.3
200_200_0.10_0.75	best	12,522	11,656	11,846	11,191	11,535	12,271	11,972	**12,522**	**12,522**	**12,522**	12,350	**12,522**	**12,522**
	Avg		10,888.7	11,194.3	10,424.1	10,969.4	11,430.23	11,232.55	11,898.73	11,586.26	12,171.6	11,852.44	11,906.97	**12,330.6**
	std dev		237.85	249.58	197.88	302.52	403.33	349.39	391.83	419.09	220.68	371.57	382.91	101.1
200_200_0.15_0.85	best	12,317	11,792	11,521	11,287	11,469	11,804	12,167	**12,317**	11,911	12,187	11,993	**12,317**	**12,317**
	Avg		10,827.5	10,945	10,345.9	10,717.1	11,062.06	11,026.81	11,584.64	11,288.25	11,746	11,612.07	11,594.9	**11,975.1**

Table 6. *Cont.*

Instance	Results	Best Known	GA	BABC	ABC_{bin}	binDE	bWSA	gPSO*	gPSO	intAgents	DH-jaya	GWOfbd	GWOrbd	MLCSBO
	std dev		334.43	255.14	273.47	341.08	423.9	421.22	275.32	410.54	181.18	217.13	301.1	140.4
300_300_0.10_0.75	best	12,817	12,055	12,186	11,494	12,304	12,644	12,736	12,695	12,695	12,695	12,784	12,695	**12,817**
	Avg		11,755.1	11,945.8	10,922.3	11,864.4	12,227.56	11,934.64	12,411.27	12,310.19	12,569.3	12,441	12,446.21	**12,716.4**
	std dev		144.45	127.8	182.63	160.42	308.11	293.83	225.8	238.32	114.13	247.67	227.47	69.2
300_300_0.15_0.85	best	11,585	10,666	10,382	9633	10,382	11,113	10,724	**11,425**	**11,425**	11,113	**11,425**	**11,425**	**11,425**
	Avg		10,099.2	9859.69	9186.87	9710.37	10,216.71	9906.81	10,568.41	10,384	10,701.9	10,632.71	10,648.53	**11,408.8**
	std dev		337.42	177.02	147.78	208.48	351.12	399.13	327.48	378.42	153.66	345.63	328.13	26.7
400_400_0.10_0.75	best	11,665	10,570	10,626	10,160	10,462	11,199	11,048	11,531	11,531	11,310	11,531	11,531	**11,665**
	Avg		10,112.4	10,101.1	9549.04	9975.8	10,624.79	10,399.97	10,958.96	10,756.92	10,914.8	10,961.25	10,964.98	**11,600.9**
	std dev		157.89	196.99	141.27	185.57	266.46	281.99	274.9	250.56	216.47	258.47	276.16	73.9
400_400_0.15_0.85	best	11,325	9235	9541	9033	9388	10,915	10,264	10,927	10,927	10,915	10,927	10,927	**11,325**
	Avg		8793.76	9032.95	8365.62	8768.42	9580.64	9195.24	9845.17	9608.07	9969.9	9849.04	9873.27	**11,271.2**
	std dev		169.52	194.18	153.4	212.24	411.83	311.9	358.91	363.72	287.61	343.9	373.7	62.4
500_500_0.10_0.75	best	11,249	10,460	10,755	10,071	10,546	10,827	10,647	10,888	10,960	10,960	10,921	10,960	**11,078**
	Avg		10,185.4	10,328.5	9738.17	10,227.7	10,482.8	10,205.08	10,681.46	10,610.53	10,703.5	10,716.55	10,742.98	**10,954.3**
	std dev		114.19	91.62	111.63	103.32	165.62	190.05	125.36	169.73	105.18	140.87	130.05	74.2
500_500_0.15_0.85	best	10,381	9496	9318	9262	9312	10,082	9839	10,194	**10,381**	10,176	10,194	10,194	10,209
	Avg		8882.88	9180.74	8617.91	9096.13	9478.71	9106.64	9703.62	9578.89	9801.5	9758.61	9737.48	**10,056.6**
	std dev		158.21	84.91	141.32	145.45	262.44	257.65	252.84	278.06	222.21	243.59	272.51	107.2

Table 7. Comparison between GA, BABC, ABC bin, gPSO*, gPSO, intAgents, DH-Jaya, GWOfbd, GWOrbd and MLCSBO algorithms for medium instances.

Instance	Results	Best Known	GA	BABC	ABCbin	binDE	bWSA	gPSO*	gPSO	intAgents	DH-jaya	GWOfbd	GWOrbd	MLCSBO
85_100_0.10_0.75	best	12,045	11,454	11,664	11,206	11,352	11,947	11,710	**12,045**	**12,045**	**12,045**	**12,045**	**12,045**	**12,045**
	Avg		11,092.7	11,182.7	10,879.5	11,075	11,233.16	11,237.05	11,486.95	11,419.75	11,570.6	11,441.23	11,430.44	**11,945.5**
	std dev		171.22	183.57	163.62	119.42	216.67	168.96	137.52	140.77	177.86	11172	127.56	126.6
85_100_0.15_0.85	best	12,369	12,124	**12,369**	12,006	**12,369**	**12,369**	**12,369**	**12,369**	**12,369**	**12,369**	**12,369**	**12,369**	**12,369**
	Avg		11,326.3	12,081.6	11,485.3	11,875.9	11,342.7	11,684.46	11,994.36	11,885.21	12,318	11,917.83	11,942.93	**12,253.1**
	std dev		417	193.79	248.33	336.94	474.76	353.79	436.81	431.67	181.92	442.25	418.72	81.6
185_200_0.10_0.75	best	13,696	12,841	13,047	12,308	13,024	13,505	13,298	**13,696**	**13,696**	**13,696**	13,647	**13,696**	**13,696**
	Avg		12,236.6	12,522.8	11,667.9	12,277.5	12,689.09	12,514.72	13,204.26	13,084.52	13,350.2	13,121.23	13,125.85	**13,651.7**
	std dev		198.18	201.35	177.14	234.24	336.51	356.2	366.56	388.39	182.56	365.41	367.06	36.1
185_200_0.15_0.85	best	11,298	10,920	10,602	10,376	10,547	10,831	10,856	**11,298**	**11,298**	**11,298**	**11,298**	**11,298**	**11,298**
	Avg		10,351.5	10,150.6	9684.33	10,085.4	10,228.07	10,208.33	10,801.41	10,780.14	10,828.9	10,871.49	10,819.34	**11,068.7**
	std dev		208.08	152.91	184.84	160.6	286.92	263.73	205.76	239.61	191.76	240.32	239.52	162.4
285_300_0.10_0.75	best	11,568	10,994	11,158	10,269	11,152	**11,568**	11,310	**11,568**	**11,568**	**11,568**	**11,568**	**11,568**	**11,568**
	Avg		10,640.1	10,775.9	9957.09	10,661.3	11,105.09	10,761.96	11,317.99	11,205.72	11,327.7	10,001.33	10,014.47	**11,546.0**
	std dev		126.84	116.8	141.48	149.84	197.78	199.43	182.82	258.49	166.91	174.62	215.35	13.5
285_300_0.15_0.85	best	11,802	11,093	10,528	10,051	10,528	11,377	11,226	11,517	11,517	11,401	11,590	**11,763**	**11,763**
	Avg		10,190.3	9897.92	9424.15	9832.32	10,452.03	10,309.19	10,899.2	10,747.33	11,025.9	10,470.33	10,871.49	**11,564.3**
	std dev		249.76	186.53	197.14	232.72	416.76	389.12	30036	334.25	208.08	340.47	332.09	129.2
385_400_0.10_0.75	best	10,600	9799	10,085	9235	9883	10,414	9871	10,483	10,326	10,414	10,397	10,483	**10,600**
	Avg		9432.82	9537.5	8904.94	9314.57	9778.03	9552.14	10,013.43	9892.17	10,017	10,043.23	9902.72	**10,400.7**
	std dev		163.84	184.62	111.85	191.59	221.49	234.1	202.4	179.19	141.15	163.95	180.32	47.3
385_400_0.15_0.85	best	10,506	9173	9456	8932	9352	10,077	9389	10,338	10,131	10,302	10,302	10,302	**10,506**
	Avg		8703.66	9090.03	8407.06	8846.99	9203.52	8881.17	9524.98	9339.67	9565.72	9472.39	9455.24	**10,162.0**
	std dev		154.15	156.69	148.52	210.91	303.12	283.3	286.16	288.88	237.9	242.25	261.75	172.7
485_500_0.10_0.75	best	11,321	10,311	10,823	10,357	10,728	10,835	10,595	11,094	11,094	10,971	10,989	11,097	**11,125**
	Avg		9993.16	10,483.4	9615.37	10,159.4	10,607.21	10,145.26	10,687.62	10,603.53	10,754.8	10,702.72	10,725.26	**10,965.3**
	std dev		117.73	228.34	151.41	198.49	191.86	199.99	168.06	204.99	112.69	154.92	160.001	96.2
485_500_0.15_0.85	best	10,220	9329	9333	8799	9218	9603	9807	10,104	10,104	9715	10,104	10,104	**10,220**
	Avg		8849.46	9085.57	8347.82	8919.64	9141.94	8917.44	9383.28	9259.36	9467.8	9462	9455.24	**10,095.7**
	std dev		141.84	115.62	122.65	168.9	180.42	267.49	241.01	268.33	106.55	229.88	261.74	74.1

Table 8. Comparison between BABC, DH-Jaya, and MLBO algorithms for large instances.

		BABC				DH-Jaya				MLBO			
Instance	**Best Known**	**Best**	**Avg**	**Std**	t_{avg}	**Best**	**Avg**	**Std**	t_{avg}	**Best**	**Avg**	**Std**	t_{avg}
600_585_0.10_0.75	9914	9098	9026.1	34.9	498.6	9640	9450.0	60.2	690.5	**9721**	**9668.6**	54.2	88.3
600_585_0.15_0.85	9357	8736	8540.5	20.5	172.5	9187	8998.5	79.2	881.3	**9313**	**9045.9**	119.6	320.5
700_685_0.10_0.75	9881	9311	9176.3	46.9	363.4	**9790**	**9602.0**	56.0	543.2	9736	9545.7	103.3	162.5
700_685_0.15_0.85	9163	8671	8397.4	87.7	302.6	9106	**8894.1**	140.5	426.1	**9135**	8834.1	106.7	356.7
800_785_0.10_0.75	9837	9275	9192.4	20.3	253.3	**9771**	**9540.1**	48.0	637.3	9470	9268	101.6	316.6
800_785_0.15_0.85	9024	8447	8366.5	72.0	254.3	**8797**	**8649.0**	63.0	236.8	**8907**	8611.9	66.3	200.9
900_885_0.10_0.75	9725	8953	8837.2	103.2	471.4	**9455**	**9249.5**	109.1	687.2	9454	9142	116.9	260.1
900_885_0.15_0.85	8620	8072	7881.2	88.5	228.4	**8418**	**8244.5**	87.9	316.6	8427	8120.5	186.2	264.3
1000_985_0.10_0.75	9668	9276	9254.2	27.9	640.5	**9424**	**9306.9**	45.0	309.9	9146	8642.4	242.3	202.5
1000_985_0.15_0.85	8453	8133	8099.1	25.4	648.2	**8433**	**8280.5**	90.9	312.6	8149	7755.7	223.7	234.0
600_600_0.10_0.75	10,524	10,207	9939.4	47.5	66.7	**10,507**	**10,504.3**	19.7	321.2	**10,518**	10,470.9	22.5	192.4
600_600_0.15_0.85	9062	8621	8361.8	101.3	455.5	8910	8785.6	43.5	572.0	**8939**	**8891.3**	31.7	570.6
700_700_0.10_0.75	9786	9078	9056.5	21.9	224.4	9512	9409.0	28.7	809.8	**9786**	**9416.6**	156.6	302.3
700_700_0.15_0.85	9229	8614	8290.2	77.6	126.8	**9121**	**8985.5**	65.9	507.7	9068	8786	140.9	244.1
800_800_0.10_0.75	9932	9517	9305.4	56.8	418.5	**9890**	**9656.4**	51.4	567.1	9679	9458.9	110.7	278.8
800_800_0.15_0.85	9101	8444	8163.8	132.7	376.7	**8961**	**8774.2**	59.8	161.7	8864	8433.7	175.9	276.7
900_900_0.10_0.75	9745	9290	9273.0	14.6	460.0	9526	**9462.9**	37.8	671.0	**9533**	9289.8	138.6	254.4
900_900_0.15_0.85	8990	8118	8114.5	9.2	151.0	**8718**	**8492.9**	62.3	702.7	8647	8233.4	279.8	250.5
1000_1000_0.10_0.75	9544	9030	8891.3	39.0	658.0	**9348**	**9250.8**	53.7	542.2	9062	8656.8	196.8	198.9
1000_1000_0.15_0.85	8474	7867	7627.8	44.9	635.0	**8330**	**8037.9**	71.9	932.6	8106	7767.3	189.1	318.6
585_600_0.10_0.75	10,393	9768	9677.8	81.9	535.9	**10,300**	**10,161.5**	72.8	98.2	10,001	9954	52.1	160.0
585_600_0.15_0.85	9256	8689	8623.8	28.5	461.9	9031	**8944.2**	61.7	616.6	**9256**	8921.7	122.6	246.4
685_700_0.10_0.75	10,121	9796	9627.4	73.2	248.7	**10,070**	**9953.6**	49.0	430.2	9914	9633.3	144.9	210.7
685_700_0.15_0.85	9176	8453	8424.9	4.8	958.7	9102	**8860.8**	106.4	160.0	**9110**	8828.6	85.9	282.2
785_800_0.10_0.75	9384	8765	8658.5	54.3	869.0	**9123**	**8885.1**	54.1	316.5	9039	8875.4	87.4	304.4
785_800_0.15_0.85	8746	8249	8021.9	117.1	577.0	**8556**	**8482.3**	51.5	604.6	8555	8280.5	108.4	310.1
885_900_0.10_0.75	9318	8938	8897.6	30.2	587.2	**9137**	**9079.1**	46.7	590.4	9019	8761.6	114.9	186.5
885_900_0.15_0.85	8425	7610	7518.0	50.5	869.7	**8217**	**7881.4**	65.8	140.9	8001	7766.7	134.6	326.2
985_1000_0.10_0.75	9193	8914	8741.3	101.8	739.9	**9067**	**8994.5**	45.0	313.1	8934	8435.7	190.5	148.1
985_1000_0.15_0.85	8528	8071	8066.5	15.2	486.5	**8453**	**8425.3**	48.7	504.0	8114	7600.7	287.6	190.4
Average	9352.3	8800.4	8668.4	54.3	458.0	9196.7	9041.4	62.5	486.8	9120.1	8836.6	136.4	255.3
p-value		3.5×10^{-6}	0.008			0.02	5.7×10^{-5}						

5. Conclusions

In this research, a hybrid k-means cuckoo search algorithm has been proposed. This hybrid binarization method applies the k-means technique to binarize the solutions generated by the cuckoo search algorithm. Additionally, in order for the procedure to be efficient, it was reinforced with a greedy initialization algorithm and with a local search operator. The proposed hybrid technique was used to solve cases of the set-union knapsack problem on a medium and large scale. The role of binarization and local search operators was investigated. To do this, a random operator was designed using two transition probabilities Random03 and Random05 , which were compared in different situations. Finally, when the proposed approach is compared with several state-of-the-art methods, it is observed that the proposed algorithm is capable of improving the previous results in most cases. We highlight that the proposed algorithm uses a general binarization framework based on k-means and which can be easily adapt different metaheuristics and integrate with initiation and local search operators and in this particular case solve SUKP giving reasonable results.

According to the behavior of MLCSBO, it is observed that in the first 30 instances, it performed robustly, significantly outperforming the algorithms used in the comparison. However, in the 30 largest instances, their efficiency was not as clear when compared to the algorithms that had solved these instances. When it came to increasing the exploitation capacity of the local search operator, increasing T, there were no improvements. The above suggests three ideas for new lines of research. The first idea is to improve the search space exploration, this can be achieved using different solution initiation mechanisms, in MLBO, a greedy initialization operator was used. The second idea, thinking that the algorithm could be trapped in local optimum, the incorporation of a perturbation operator can be investigated. At this point, it can also consider the use of machine learning techniques such as the k-nearest neighborhood. Finally, the last idea aims to explore other binarization techniques based on other clustering algorithms or some other binarization strategies.

Author Contributions: J.G.: Conceptualization, investigation, methodology, writing—review and editing, project administration, resources, formal analysis. J.L.-R., M.B.-R., F.A.: Conceptualization, investigation, validation. B.C., R.S., J.-M.R., P.M., A.P.B., A.P.F., G.A.: Validation, funding acquisition. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by: José García was supported by the Grant CONICYT/FONDECYT/INICIACION/ 11180056. PROYECTO DI INVESTIGACIÓN INNOVADORA INTERDISCIPLINARIA: 039.414/2021. José Lemus-Romani is supported by National Agency for Research and Development (ANID)/ Scholarship Program/DOCTORADO NACIONAL/2019-21191692. Marcelo Becerra-Rozas is supported by National Agency for Research and Development (ANID)/Scholarship Program/DOCTORADO NACIONAL/2021-21210740. Broderick Crawford is supported by Grant CONICYT / FONDECYT/REGULAR/1210810.Ricardo Soto is supported by Grant CONICYT/FONDECYT/REGULAR/1190129. Broderick Crawford, Ricardo Soto, and Marcelo Becerra-Rozas are supported by Grant Nucleo de Investigacion en Data Analytics/VRIEA/PUCV/039.432/2020.

Institutional Review Board Statement: Not applicable for studies not involving humans or animals.

Informed Consent Statement: Informed consent was obtained from all subjects involved in the study.

Data Availability Statement: The data set used in this article can be obtained from: https://drive.google.com/drive/folders/1aH11zXXBFtWbKjS9MlKxv-7eZjgcvCpL?usp=sharing, accessed on 14 October 2021. The results of the experiments are in: https://drive.google.com/drive/u/2/folders/1xLY1Cu8loizh44oVa7vS0s4nqUAvhHNV, accessed on 14 October 2021.

Acknowledgments: José García was supported by the Grant CONICYT/FONDECYT/INICIACION/11180056. PROYECTO DI INVESTIGACIÓN INNOVADORA INTERDISCIPLINARIA: 039.414/2021. José Lemus-Romani is supported by National Agency for Research and Development (ANID)/ Scholarship Program/DOCTORADO NACIONAL/2019-21191692. Marcelo Becerra-Rozas is supported by National Agency for Research and Development (ANID)/Scholarship Program/DOCTORADO NACIONAL/2021-21210740. Broderick Crawford is supported by Grant CONICYT/FONDECYT/REGULAR/1210810.Ricardo Soto is supported by Grant CONICYT/FONDECYT/REGULAR/1190129.

Broderick Crawford, Ricardo Soto, and Marcelo Becerra-Rozas are supported by Grant Nucleo de Investigacion en Data Analytics/VRIEA/PUCV/039.432/2020.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Guo, H.; Liu, B.; Cai, D.; Lu, T. Predicting protein–protein interaction sites using modified support vector machine. *Int. J. Mach. Learn. Cybern.* **2018**, *9*, 393–398. [CrossRef]
2. Korkmaz, S.; Babalik, A.; Kiran, M.S. An artificial algae algorithm for solving binary optimization problems. *Int. J. Mach. Learn. Cybern.* **2018**, *9*, 1233–1247. [CrossRef]
3. Penadés-Plà, V.; García-Segura, T.; Yepes, V. Robust design optimization for low-cost concrete box-girder bridge. *Mathematics* **2020**, *8*, 398. [CrossRef]
4. Al-Madi, N.; Faris, H.; Mirjalili, S. Binary multi-verse optimization algorithm for global optimization and discrete problems. *Int. J. Mach. Learn. Cybern.* **2019**, *10*, 3445–3465. [CrossRef]
5. Talbi, E.G. Combining metaheuristics with mathematical programming, constraint programming and machine learning. *Ann. Oper. Res.* **2016**, *240*, 171–215. [CrossRef]
6. Tsao, Y.C.; Vu, T.L.; Liao, L.W. Hybrid Heuristics for the Cut Ordering Planning Problem in Apparel Industry. *Comput. Ind. Eng.* **2020**, *144*, 106478. [CrossRef]
7. Chhabra, A.; Singh, G.; Kahlon, K.S. Performance-aware energy-efficient parallel job scheduling in HPC grid using nature-inspired hybrid meta-heuristics. *J. Ambient. Intell. Humaniz. Comput.* **2021**, *12*, 1801–1835. [CrossRef]
8. Caserta, M.; Voß, S. Metaheuristics: Intelligent problem solving. In *Matheuristics*; Springer: Berlin, Germany, 2009; pp. 1–38.
9. Schermer, D.; Moeini, M.; Wendt, O. A matheuristic for the vehicle routing problem with drones and its variants. *Transp. Res. Part Emerg. Technol.* **2019**, *106*, 166–204. [CrossRef]
10. Roshani, M.; Phan, G.; Roshani, G.H.; Hanus, R.; Nazemi, B.; Corniani, E.; Nazemi, E. Combination of X-ray tube and GMDH neural network as a nondestructive and potential technique for measuring characteristics of gas-oil–water three phase flows. *Measurement* **2021**, *168*, 108427. [CrossRef]
11. Roshani, S.; Jamshidi, M.B.; Mohebi, F.; Roshani, S. Design and Modeling of a Compact Power Divider with Squared Resonators Using Artificial Intelligence. *Wirel. Pers. Commun.* **2021**, *117*, 2085–2096. [CrossRef]
12. Nazemi, B.; Rafiean, M. Forecasting house prices in Iran using GMDH. *Int. J. Hous. Mark. Anal.* **2020**, *14*, 555–568. [CrossRef]
13. Talbi, E.G. Machine Learning into Metaheuristics: A Survey and Taxonomy. *ACM Comput. Surv. (CSUR)* **2021**, *54*, 1–32.
14. Calvet, L.; de Armas, J.; Masip, D.; Juan, A.A. Learnheuristics: Hybridizing metaheuristics with machine learning for optimization with dynamic inputs. *Open Math.* **2017**, *15*, 261–280. [CrossRef]
15. Crawford, B.; Soto, R.; Astorga, G.; García, J.; Castro, C.; Paredes, F. Putting continuous metaheuristics to work in binary search spaces. *Complexity* **2017**, *2017*, 8404231. [CrossRef]
16. García, J.; Lalla Ruiz, E.; Voß, S.; Lopez Droguett, E. Enhancing a machine learning binarization framework by perturbation operators: Analysis on the multidimensional knapsack problem. *Int. J. Mach. Learn. Cybern.* **2020**, *11*, 1951–1970. [CrossRef]
17. García, J.; Astorga, G.; Yepes, V. An analysis of a KNN perturbation operator: An application to the binarization of continuous metaheuristics. *Mathematics* **2021**, *9*, 225. [CrossRef]
18. García, J.; Martí, J.V.; Yepes, V. The buttressed walls problem: An application of a hybrid clustering particle swarm optimization algorithm. *Mathematics* **2020**, *8*, 862. [CrossRef]
19. García, J.; Yepes, V.; Martí, J.V. A hybrid k-means cuckoo search algorithm applied to the counterfort retaining walls problem. *Mathematics* **2020**, *8*, 555. [CrossRef]
20. Goldschmidt, O.; Nehme, D.; Yu, G. Note: On the set-union knapsack problem. *Nav. Res. Logist.* **1994**, *41*, 833–842. [CrossRef]
21. Wei, Z.; Hao, J.K. Multistart solution-based tabu search for the Set-Union Knapsack Problem. *Appl. Soft Comput.* **2021**, *105*, 107260. [CrossRef]
22. Ozsoydan, F.B.; Baykasoglu, A. A swarm intelligence-based algorithm for the set-union knapsack problem. *Future Gener. Comput. Syst.* **2019**, *93*, 560–569. [CrossRef]
23. Liu, X.J.; He, Y.C. Estimation of distribution algorithm based on Lévy flight for solving the set-union knapsack problem. *IEEE Access* **2019**, *7*, 132217–132227. [CrossRef]
24. Tu, M.; Xiao, L. System resilience enhancement through modularization for large scale cyber systems. In Proceedings of the 2016 IEEE/CIC International Conference on Communications in China (ICCC Workshops), Chengdu, China, 27–29 July 2016; pp. 1–6, 27–29.
25. Yang, X.; Vernitski, A.; Carrea, L. An approximate dynamic programming approach for improving accuracy of lossy data compression by Bloom filters. *Eur. J. Oper. Res.* **2016**, *252*, 985–994. [CrossRef]
26. Feng, Y.; An, H.; Gao, X. The importance of transfer function in solving set-union knapsack problem based on discrete moth search algorithm. *Mathematics* **2019**, *7*, 17. [CrossRef]
27. Wei, Z.; Hao, J.K. Kernel based tabu search for the Set-union Knapsack Problem. *Expert Syst. Appl.* **2021**, *165*, 113802. [CrossRef]
28. García, J.; Crawford, B.; Soto, R.; Castro, C.; Paredes, F. A k-means binarization framework applied to multidimensional knapsack problem. *Appl. Intell.* **2018**, *48*, 357–380. [CrossRef]

29. Lister, W.; Laycock, R.; Day, A. *A Key-Pose Caching System for Rendering an Animated Crowd in Real-Time*; Computer Graphics Forum; Wiley Online Library: Hoboken, NJ, USA, 2010; Volume 29, pp. 2304–2312.
30. Arulselvan, A. A note on the set union knapsack problem. *Discret. Appl. Math.* **2014**, *169*, 214–218. [CrossRef]
31. Wei, Z.; Hao, J.K. Iterated two-phase local search for the Set-Union Knapsack Problem. *Future Gener. Comput. Syst.* **2019**, *101*, 1005–1017. [CrossRef]
32. He, Y.; Xie, H.; Wong, T.L.; Wang, X. A novel binary artificial bee colony algorithm for the set-union knapsack problem. *Future Gener. Comput. Syst.* **2018**, *78*, 77–86. [CrossRef]
33. Feng, Y.; Yi, J.H.; Wang, G.G. Enhanced moth search algorithm for the set-union knapsack problems. *IEEE Access* **2019**, *7*, 173774–173785. [CrossRef]
34. Wu, C.; He, Y. Solving the set-union knapsack problem by a novel hybrid Jaya algorithm. *Soft Comput.* **2020**, *24*, 1883–1902. [CrossRef]
35. Zhou, Y.; Zhao, M.; Fan, M.; Wang, Y.; Wang, J. An efficient local search for large-scale set-union knapsack problem. *Data Technol. Appl.* **2020**.
36. Gölcük, İ.; Ozsoydan, F.B. Evolutionary and adaptive inheritance enhanced Grey Wolf Optimization algorithm for binary domains. *Knowl.-Based Syst.* **2020**, *194*, 105586. [CrossRef]
37. Crawford, B.; Soto, R.; Lemus-Romani, J.; Becerra-Rozas, M.; Lanza-Gutiérrez, J.M.; Caballé, N.; Castillo, M.; Tapia, D.; Cisternas-Caneo, F.; García, J.; et al. Q-Learnheuristics: Towards Data-Driven Balanced Metaheuristics. *Mathematics* **2021**, *9*, 1839. [CrossRef]
38. Lanza-Gutierrez, J.M.; Crawford, B.; Soto, R.; Berrios, N.; Gomez-Pulido, J.A.; Paredes, F. Analyzing the effects of binarization techniques when solving the set covering problem through swarm optimization. *Expert Syst. Appl.* **2017**, *70*, 67–82. [CrossRef]
39. He, Y.; Wang, X. Group theory-based optimization algorithm for solving knapsack problems. *Knowl.-Based Syst.* **2021**, *219*, 104445. [CrossRef]
40. García, J.; Moraga, P.; Valenzuela, M.; Pinto, H. A db-scan hybrid algorithm: an application to the multidimensional knapsack problem. *Mathematics* **2020**, *8*, 507. [CrossRef]
41. Baykasoğlu, A.; Ozsoydan, F.B.; Senol, M.E. Weighted superposition attraction algorithm for binary optimization problems. *Oper. Res.* **2020**, *20*, 2555–2581. [CrossRef]
42. Ozsoydan, F.B. Artificial search agents with cognitive intelligence for binary optimization problems. *Comput. Ind. Eng.* **2019**, *136*, 18–30. [CrossRef]