Int. J. Adv. Eng. Pure Sci. 2021, 33(3): 406-414
DOI: 10.7240/jeps.800056

RESEARCH ARTICLE / ARAŞTIRMA MAKALESİ

Pool-based Evolutionary Algorithm for the Bin Packing Problem

Kutu Paketleme Problemi için Havuz Tabanlı Evrimsel Algoritma

Tuğba Zeynep YILDIZ[1], **Betül BOZ**[1]

[1] *Computer Engineering Department Marmara University, 34722, Istanbul, Turkey*

Abstract
Bin packing problem is one of the most important optimization problems from the literature. In this work, we propose a novel pool-based evolutionary algorithm for solving the one-dimensional bin packing problem. The algorithm uses the pool-based crossover operator that aims to increase the search space of the problem and combine and remap method as a local search technique that aims to improve the quality of the solution by considering underutilized bins in the offspring. In our experimental study, the performance of the proposed method is compared with six algorithms from the literature using medium and hard instances in the benchmark problem sets. As a result, the proposed study performs better than the algorithms in the literature in 13% of medium instances and 80% of hard instances.
Keywords: Bin packing problem, evolutionary algorithms, crossover operator, problem specific operator design.

Öz
Kutu paketleme problemi literatürdeki en önemli optimizasyon problemlerinden biridir. Bu çalışmada, tek boyutlu kutu paketleme probleminin çözümü için havuz tabanlı evrimsel algoritma öneriyoruz. Algoritma, problemin arama alanını arttırmayı amaçlayan havuz tabanlı bir çaprazlama operatöründen ve yavru çözümdeki tamamen kullanılmayan kutuları dikkate alarak çözümün kalitesini iyileştirmeyi amaçlayan birleştirmeyi ve tekrar atamayı sağlayan yerel bir arama tekniğinden yararlanmaktadır. Deneysel çalışmamızda önerdiğimiz yöntemin performansı, literatürde bulunan altı algoritma ile kıyaslama problem setlerinde bulunan orta ve zor örnekler kullanılarak karşılaştırılmıştır. Sonuç olarak önerdiğimiz çalışma, orta örneklerin %13'ünde ve zor örneklerin %80'inde literatürdeki algoritmalardan daha iyi performans göstermektedir.
Anahtar Kelimeler: Kutu paketleme problemi, evrimsel algoritmalar, çaprazlama operatörü, probleme özgü operator tasarımı.

I. INTRODUCTION

Bin packing problem is a well-known optimization problem which can be applied to many real-life problems including industrial and logistic applications, multiprocessor scheduling and cloud computing. Given a set of items with different weights and an unlimited number of bins with fixed bin capacity, the objective of the problem is to pack these items to minimum number of bins such that the total weight of the items assigned to a bin does not exceed the capacity of the bin. Bin packing problem has various versions such as one-dimensional packing, two-dimensional packing, three-dimensional packing, regular or irregular packing, packing by cost or packing by weight. In this study, we consider one dimensional packing by weight and our objective is to minimize the total number of bins used.

Bin-packing is an NP-Complete [1] problem and many heuristics and meta-heuristics have been proposed for the solution of the problem. The First Fit algorithm [2], the Best Fit algorithm [3], the Next Fit algorithm [2] and graph-based algorithm [4] are examples for the heuristic solutions of the problem. A variety of meta heuristics such as ant colony optimization [5, 6], cuckoo search algorithm [7, 8], firefly algorithm [9] and whale optimization algorithm [10], genetic algorithms [11, 12], simulated annealing [13] have also been used to solve the problem.

Firefly colony optimization algorithm (FCO) [5] is a greedy metaheuristic using positive feedback to avoid convergence to low quality solutions. Ant system algorithm (AS) [6] combines ant colony algorithm with a local search technique. Adaptive Cuckoo Search Algorithm (ACS) [7] combines cuckoo search algorithm with Ranked-Order-Value (ROV) technique as a decoding mechanism to obtain discrete solutions. Quantum Inspired Cuckoo Search Algorithm (QICS) [8] defines the solutions using quantum representation based on qubit representation and uses a novel hybrid quantum measure operation inspired by the first fit heuristic. Firefly algorithm (FA) [9] uses L´evy flights as a search strategy which enables the algorithm to converge quickly. Improved Lévy-based whale optimization algorithm (ILWOA) [10] uses Lévy flight for whale movements to improve the exploration capabilities of whale optimization algorithm. It also uses a logistic chaotic map to switch between exploration and exploitation. These algorithms are the reference studies used in the experiments.

Corresponding Author: Betül BOZ, Tel: 0 (216) 777 35 34, e-posta: betul.demiroz@marmara.edu.tr
Submitted: 25.09.2020, **Revised:** 24.05.2021, **Accepted:** 24.05.2021

Evolutionary algorithms are heuristic approaches that can be used to find solutions to NP-Complete problems. Basically evolutionary algorithms mimic the nature and are based on the idea of the survival of the fittest. In this study, we propose a pool-based evolutionary algorithm (PBEA) for the solution of bin packing problem using a problem-specific crossover operator and a local search technique. The main contributions of this study are: (1) the pool-based crossover operator that aims to increase the diversity of the solution; (2) the combine and remap local search technique that aims to increase the utilization of the bins; (3) intelligent packing by rearranging items in underutilized bins, therefore increase performance and decrease bin usage. Our experimental study indicates that it outperforms related studies from the literature for medium and hard class instances.

Our paper is organized as follows: Bin Packing problem is defined in the next section. In Section III, the details of the proposed algorithm are given using convenient examples. The performance of our algorithm and its comparison with six algorithms from the literature are shown in Section IV. Finally, we discuss our contributions and give future directions for the problem in Section V.

II. PROBLEM DEFINITION

Given n items having different weights, and bin capacities, the objective of the bin packing problem is to assign all items to the minimum number of bins in which the total weight of items assigned to a bin does not exceed the capacity of the bin. The problem can be formulated as follows:

$$minimize\ z(y) = \sum_{i=1}^{n} y_i \quad (1)$$

Subject to constraints:

$$\sum_{i=1}^{n} x_{ij} = 1, j = 1..n \quad (2)$$

$$\sum_{i=1}^{n} w_i\, x_{ij} \leq C_j, j = 1..n \quad (3)$$

$$y_i \in \{0,1\}, \mathrm{i} = 1..n \quad (4)$$

$$x_{ij} \in \{0,1\}, \mathrm{i} = 1..n, j = 1..n \quad (5)$$

The variable y_j is used to indicate whether bins are used. If bin j is used, y_j is equal to 1, but if it is empty, y_j is set to 0. If item i is placed to bin j, x_{ij} is set to 1, otherwise 0. The objective of the problem as represented in Eq.1 is to minimize the total number of bins used. The first constraint guarantees that each items is placed to only one bin as shown in Eq.2. The second constraint ensures that the total weight of the items assigned to a bin cannot exceed the bin capacity (Eq.3). In the proposed work, we assume that all bins have the same capacity *C*.

III. POOL-BASED EVOLUTIONARY ALGORITHM

In bin packing problem, an infinite number of bins having the same capacity and a number of items having different weights are given. In our study, the grouping method [14] is used to represent each individual as S_i where $S_i = \{B_1, B_2, ..., B_k\}$. Each partition B_i includes the set of items assigned to the bin b_i; and k the number of bins used.

Our evolutionary algorithm starts by generating the initial population which includes candidate solutions for the bin packing problem as presented in Algorithm 1 and continues for a predefined number of iterations. At each iteration, two individuals referred as *parent1* and *parent2* are selected randomly from the population. The pool-based crossover operator and the combine and remap local search technique are applied to these parents to obtain a new offspring. The crossover operator tries to increase the search space of the solution by mixing the items assigned to the bins of the parents. The local search technique tries to decrease the total number of bins used in the offspring, so it keeps the utilized bins and combines and remaps the underutilized bins. When the local search technique is completed, the fitness value of the offspring is set to the total number of bins occupied by the offspring. If the fitness value of the offspring is better than its parents, the parent having the worst fitness value is selected for replacement.

Algorithm 1: Main scheme of the proposed Pool-based evolutionary algorithm

Input: Items *i*, number of iterations *iteration*, population size *pop_size*, bin capacity c

Output: Updated population P

1. P ← initialize_population (s, *i*, *c*)
2. **For** i=0 **to** *iteration* **do**
3. Select two parents S_1 and S_2 from P randomly
4. S_c ← crossover_operation (S_1, S_2)
5. $S_{improved}$ ← local_search (S_c)
6. P ← update_pop ()
7. **End**

3.1. Initial Population Generation

The initial population includes a predefined number of candidate solutions for the bin packing problem. At the beginning of initial population generation, all individuals are initialized to contain one bin with a given capacity. Before each individual is created, the items are shuffled and are placed to a list. Items are selected one-by-one from this list and are placed to the

first bin of the corresponding individual having enough capacity. If no such bin exists, then a new bin is created, and the item is placed to this bin. The available space of each bin is updated as new items are inserted to the bins. Once a predefined number of solutions are obtained, the initial population generation is completed.

3.2. Pool-based Crossover Operator

In this work, we use the Pool-Based Crossover (PBC) [15,16] that increases the search space while generating the bins of the offspring. It also includes a pool which contains the items that has not been assigned to any bin yet due to size constraints. The details of our crossover operator are given in Algorithm 2.

Algorithm 2: Pool-based crossover operator.

Input: First parent $S_1 = \{B_1^1, B_2^1, \ldots, B_k^1\}$, second parent $S_2 = \{B_1^2, B_2^2, \ldots, B_k^2\}$

Output: An offspring $S_0 = \{B_1, B_2, .., B_k\}$

```
Create an empty pool Pool := ⦰
i=0
While there are unselected bins in S1 or S2 do
    Create ith bin of S0 Bi: Bi := ⦰
    Set available_space(Bi) := Capacity of bins
    Select an unselected bin from S1: Bx1
    Select an unselected bin from S2: By2
    Set Bx1 and By2 as selected
    Remove items in Bx1 and By2 from S1 and S2
    Combine and shuffle items in Bx1 and By2 with the items in Pool
    While there are items in Pool do
        Select item I from Pool
        For j=0 to i do
            If weight(I) <= available_space(Bj) then
                Remove item I from Pool
                Place item I to Bj
                Update Available Space of Bj: available_space(Bj) = available_space(Bj) – weight(I)
                break
            End
        End
        Update item I: I := the next item in Pool
    End
    Increment i: i:= i +1
End
j=i
While there are items in Pool do
    Select item I from Pool
    ItemPlaced=false
    For j to i do
        If weight(I) <= available_space(Bj) then
            Remove item I from Pool
            Place item I to Bj
            Update Available Space of Bj: available_space(Bj) = available_space(Bj) – weight(I)
            ItemPlaced=true
            break
        End
    End
    If not ItemPlaced then
        Create ith bin of S0 Bi: Bi := ⦰
        Set available_space(Bi) := Capacity of bins
        Remove item I from Pool
        Place item I to Bj
        Update Available Space of Bj: available_space(Bj) = available_space(Bj) – weight(I)
        Increment i: i:= i +1
    End
    Update item I: I := the next item in Pool
End
Return offspring S0
```

Assume that two randomly selected parents represented as $S_1 = \{B_1^1, B_2^1, \ldots, B_k^1\}$ and $S_2 = \{B_1^2, B_2^2, \ldots, B_k^2\}$ with k bins are the inputs. Since parents have k partitions, the crossover operation will continue for at most k steps. The total number of bins in S_1 and S_2 does not have to be equal, and the operator will continue until it selects all items from both parents. Initially, the offspring with only one bin having an initial free space equal to the bin capacity and an empty pool is created.

At each step, one bin from S_1 and one bin from S_2 are selected randomly. These bins and the items in these bins are assigned as *selected* and will not be used by the crossover operator again. All the items that are present in the selected bins are combined with the items in the pool and are shuffled to increase the diversity of the solutions. The items in the pool are then placed into the bins of the offspring one by one. For each item x, there is a back-search operation which visits the bins currently available in the offspring one by one. Starting from the first bin B_0 to B_{i-1}, if there is any bin B_j which has free space that is equal to or greater than the weight of the item x, than this item is deleted from the pool and is placed to B_j. If no such bin can be found, item x is placed (if there is enough space) to B_i that is generated in the offspring during the current step. Once an item is placed to one of the bins in the offspring, the free space of the corresponding bin is also updated. If item x cannot be placed to any of the bins, then it is kept in the pool to be placed to one of the bins created in the next steps.

Figure 1 provides an example of the crossover operator applied to the given parents. The weights of the items used in this example are shown in Table 1. The capacity of the bins is set to 14. Assume that B_5^1 and B_3^2 are selected randomly and the items I_5 and I_1 in these bins are combined in the pool. The items in the pool are shuffled and the first bin having free space of 14 is generated as B_0. Since the weight of the first item I_5 in the pool is less than the free space of B_0, I_5 is deleted from the pool and is placed to B_0 and free space of B_0 is updated to 3. The algorithm tries to put the second item I_1 having a weight of 7 to B_0, but there is not enough space to store this item in B_0, so it is kept in the pool.

Table 1. Weights of items used in pool-based crossover operator example in Figure 1.

Item	0	1	2	3	4	5	6	7	8
Weight	5	7	3	5	12	11	10	11	9

In the second step, B_0^1 and B_4^2 are selected randomly and the items I_2, I_7 and I_8 in these bins are combined with the items in the pool. The pool is shuffled again and B_1 having a capacity of 14 is generated. The first item I_1 is placed into the newly generated bin B_1 of the offspring. The free space of B_1 is updated to 7. The weight of the second item I_2 in the pool is equal to 3, with the back-search operation, it is placed to B_0 which makes B_0 fully utilized. The algorithm tries to place items I_7 and I_8 into B_1, but the weights of these items are greater than the free space available in B_1, so they are kept in the pool. In the third step, B_3^1 and B_0^2 are selected randomly and the items I_4 and I_6 in these bins are combined with the items in the pool. After the pool is shuffled, I_4 is placed to the newly generated bin B_2. In the last step, B_2^1 and B_1^2 are selected randomly, and all items are thrown into the pool. Again, the pool is shuffled and I_3 is placed into B_1 by using the back-search operation, whereas I_6 is placed to the newly generated bin B_3.

At the end of this step, all partitions of the parents become empty, so the crossover operator considers only the items in the pool. Since a back-search operation has been performed on all items before, a new partition B_4 is generated to place these items. I_0 and I_8 is placed to B_4 and for the remaining item I_7 bin B_5 is created. The crossover operator continues until all items in the pool are placed into one of the bins.

3.3. Combine and Remap Local Search Technique

At the end of pool-based crossover operator, some bins in the offspring are fully utilized such as B_0 and B_4 as given in Figure 1 so there is no need to touch these bins. Our local search technique is called *Combine and Remap* as it tries to decrease the number of bins used in the offspring by exchanging information stored in underutilized bins as shown in Algorithm 3. This technique considers only the *underutilized bins*. This technique allows grouping of the items with lowest weight into one bin, so it increases the change of decreasing the total number of bins used in the solution. It also increases the utilization of some of the underutilized bins.

Algorithm 3: Combine and remap local search technique.

Input: An offspring $S_0 = \{B_1, B_2, .., B_k\}$

Output: The offspring $S_0 = \{B_1, B_2, .., B_k\}$

8. Create an empty pool Pool := ⦰
9. Find underutilized bins of offspring S_0
10. **While** there are underutilized bins **do**
11. Merge 2 underutilized bins to create a new partition which is utilized and add remaining items to the pool: partition, pool ← partition_reduction (underutilized bins)
12. **End**
13. Merge the pools to create new partitions
14. **Return** offspring S_0

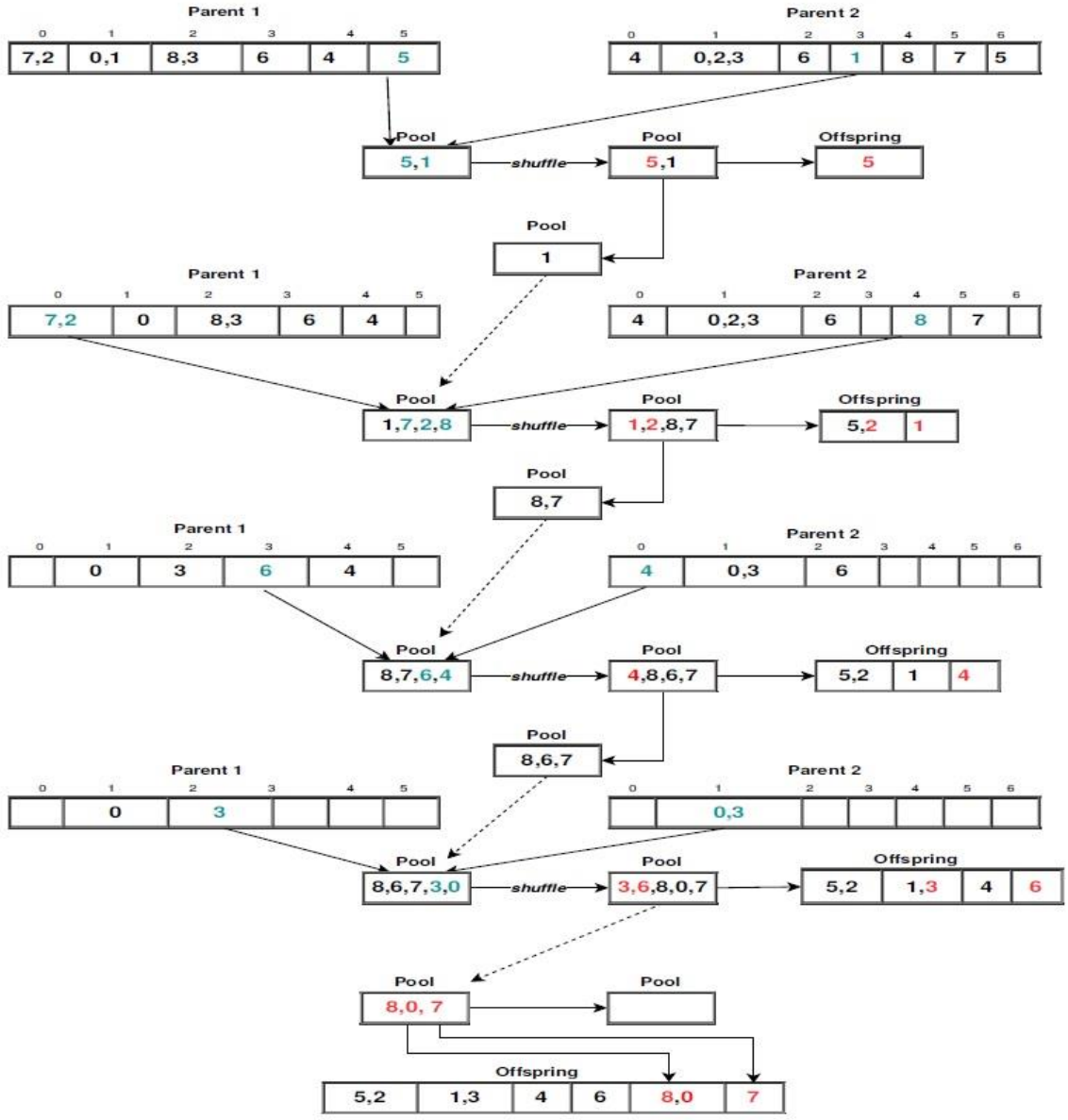

Figure 1. Pool-based crossover operator applied to the given parents to obtain an offspring.

Local search technique selects underutilized bin pairs in the offspring and combines the items in these bins to form new bins. When the items from two bins are combined, a more intelligent method than shuffling is used which places the items with the highest weight to the new bin. The items with the lowest weight are thrown to the pool to increase the chance of placing higher number of items into the same bin. The local search technique is finished when all underutilized bins are recombined and the items in the bins are rearranged. Local search technique does not always guarantee to decrease the number of bins in the offspring, but it guarantees to increase the utilization of the underutilized bins which increases the chance to find a better solution in the next iterations.

Figure 2 is an example local search scenario applied to the underutilized bins of an offspring. In this example, actual data taken from HARD0 instance of hard data sets [17] is used. The weights of the items are divided by 1000 and rounded to increase readability and are given in Table 2. The capacity of each bin is 100. This simple example shows that the local search technique can decrease the total number of bins used in the offspring by 1. The bins which have utilization less than 90% are considered in this phase. The first underutilized bin pair deleted from the offspring is B_{11} and B_{15}. The items that have the highest weight in these bins are I_5, I_{47} and I_{50} so they are used to generate the first partition P_1. The remaining items I_{124}, I_{129} and I_{186} having lower weights are thrown to the pool. Likewise, B_{20} and B_{21} are combined, P_2 is filled with items I_7, I_{41} and I_{96} having highest weight. The items with lower weights are thrown to the pool. P_3 and P_4 are generated using bin pairs B_{25} and B_{32}, B_{47} and B_{51}. All the underutilized bins have 3 items, so does the

newly generated partitions P_1 - P_4. Also, the utilization of the partitions is higher as compared to the selected bins. All remaining items that have not been placed to a partition are available in the pool and we know that the weights of these items are lower, so the pool increases the chance to generate partitions that hold more items.

Our algorithm initially selects the items with the highest weight from the pool. I_{111}, I_{121} and I_{124} are placed to P_5 and there is still 22 free space in P_5. The next item with the highest weight is I_{129} but it has a weight of 25, so it cannot be placed to this partition. Our algorithm applies best fit in this case to see if there is an item in the pool that has a weight equal to the free space available in the partition. If so, it selects that item, removes it from the pool and puts it to the partition. If there is no such item having an equal weight, then it selects the item which has a lower but closer weight. I_{186}'s weight is 21, so it is placed to P_5. Partitions P_6 and P_7 are generated using the same procedure.

The newly generated partitions P_5 and P_6 are nearly fully utilized, and all newly generated partitions have 4 items, which decreases the total number of bins in the offspring by 1. At the end of local search phase, the newly generated partitions are replaced with the underutilized bins in the offspring.

Table 2. Weights of items used in combine and remap local search technique example given in Figure 2.

I_i	5	7	19	40	41	47	54	55	57	70	93	96	111	121	124	129	132	134	148	158	159	160	186	190
W_i	35	34	33	32	31	31	30	30	30	29	28	27	26	26	26	25	25	25	24	23	23	23	21	21

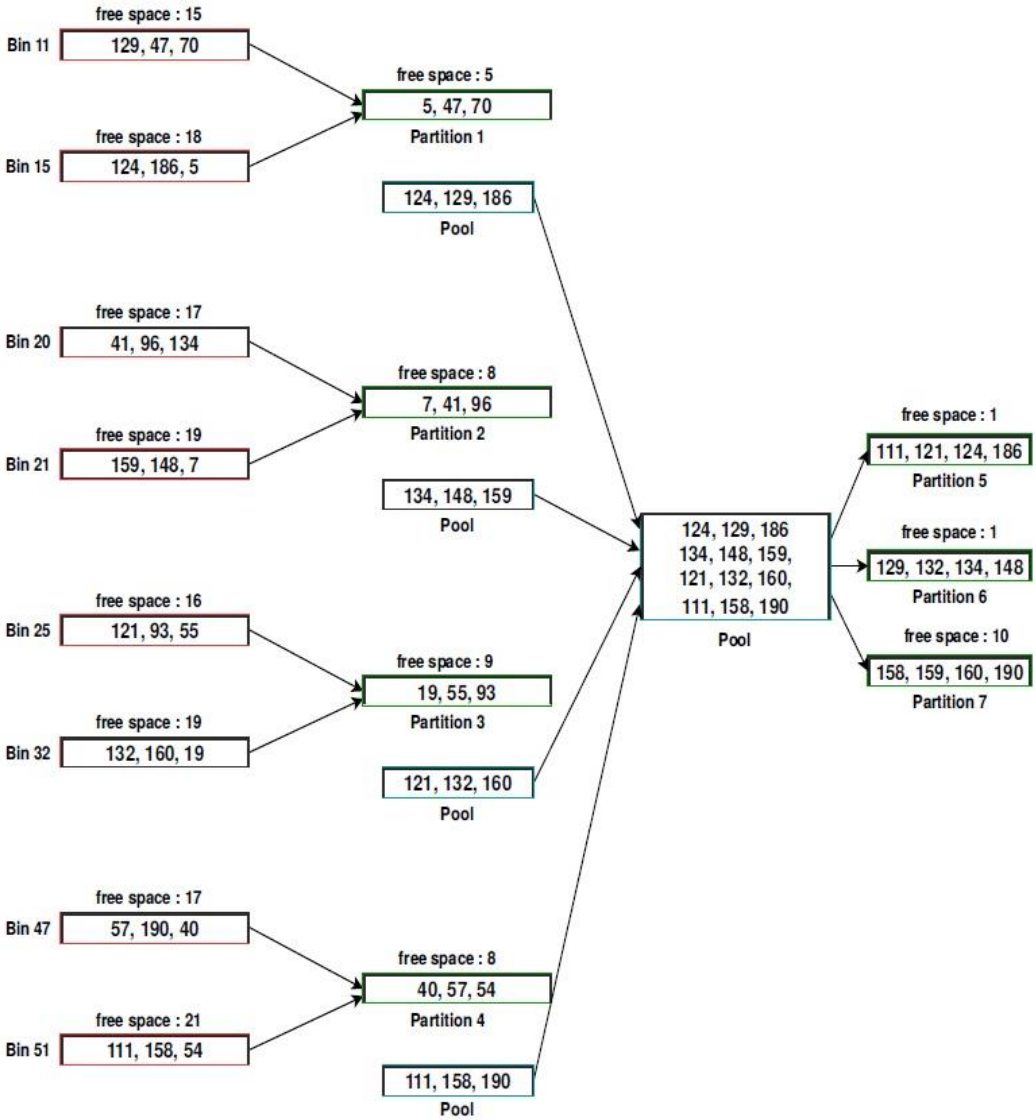

Figure 2. Applying combine and remap local search technique to the underutilized partitions of the offspring.

IV. EXPERIMENTAL EVALUATION

In this section, two set of tests are provided to measure the performance of our algorithm using medium and hard class instances selected from Scholl uniformly distributed instances [17]. First the performance of our algorithm is compared with quantum inspired cuckoo search algorithm (QICS) [8], adaptive cuckoo search algorithm (ACS) [7], firefly colony optimization algorithm (FCO) [5], firefly algorithm (FA) [9], ant system algorithm (AS) [6] and improved Lévy-based whale optimization algorithm (ILWOA) [10]. The performance of the algorithms in Table 4 and Table 6 are collected from [10], and the same instances used in this reference work are selected as medium and hard instances for the comparison of the algorithms. We also showed the performance of our algorithm using two variations of the pool-based crossover operator on hard class instances. For all the experiments, the population size and generation size are set to 100 and 500, respectively. The proposed algorithm is executed 30 times for each instance. Since the crossover operator and local search technique is always executed, both the crossover and mutation rate is 1.

The medium class instances contain 50 to 500 items whose weights are between 10-500. These items should be assigned to the minimum number of bins with a capacity of 1000. Table 3 lists the performance of our algorithm on medium class instances. The best value denotes the minimum number of bins found by the algorithm, whereas the average bin value and standard deviation are calculated using 30 runs. In 9 medium instances out of 16, our algorithm always uses the same bin number. PBEA obtains the same bin number in nearly half of the runs for 6 medium instances, whereas for 1 instance namely N4W2B1R3, we were able to obtain the best bin number 103 in 2 out of 30 runs.

The performance of the proposed work is compared with the algorithms from the literature for the same 16 medium class instances and the results are reported in Table 4. Best known column represents the best achievable bin number. The performance of the algorithms from the literature are collected from [10] and the best bin value obtained in 30 runs is given as the performance of PBEA. Our algorithm outperforms the listed algorithms in 2 instances (N4W2B1R0 and N4W2B1R3) and obtains the same performance with ACS and ILWOA in 14 instances. In 9 instances, our algorithm obtains the best-known bin value. Even if we have shown results of 16 medium class instances, indeed there are 240 instances in this class. We tested our algorithm for all 240 instances, and we are able to obtain the best-known bin value for 113 instances.

Table 3. Performance of PBEA on medium class dataset.

Instance	N	Capacity	BEST	AVG	STD
N1W1B1R2	50	1000	19	19	0
N1W1B1R9	50	1000	17	17	0
N1W1B2R0	50	1000	17	17.73	0.45
N1W1B2R1	50	1000	17	17	0
N1W1B2R3	50	1000	17	17	0
N2W1B1R0	100	1000	34	34.4	0.5
N2W1B1R1	100	1000	35	35.3	0.47
N2W1B1R3	100	1000	35	35.37	0.49
N2W1B1R4	100	1000	34	34.7	0.47
N2W3B3R7	100	1000	13	13	0
N2W4B1R0	100	1000	12	12	0
N4W2B1R0	500	1000	104	104.5	0.51
N4W2B1R3	500	1000	103	103.97	0.18
N4W3B3R7	500	1000	74	74	0
N4W4B1R0	500	1000	57	57	0
N4W4B1R1	500	1000	57	57	0

Table 4. Performance comparison of PBEA with related studies on medium class dataset.

Instance	N	Capacity	Best known	QICS	ACS	FCO	FA	AS	ILWOA	PBEA
N1W1B1R2	50	1000	19	20	19	19	20	20	19	19
N1W1B1R9	50	1000	17	18	17	17	17	17	17	17
N1W1B2R0	50	1000	17	18	17	18	18	18	17	17
N1W1B2R1	50	1000	17	17	17	17	17	18	17	17
N1W1B2R3	50	1000	16	17	17	17	17	17	17	17
N2W1B1R0	100	1000	34	36	34	35	35	37	34	34
N2W1B1R1	100	1000	34	37	35	36	36	36	35	35
N2W1B1R3	100	1000	34	37	35	36	36	36	35	35
N2W1B1R4	100	1000	34	37	34	35	35	35	34	34
N2W3B3R7	100	1000	13	13	13	13	13	13	13	13
N2W4B1R0	100	1000	12	12	12	12	12	12	12	12
N4W2B1R0	500	1000	101	109	105	106	106	107	105	**104**
N4W2B1R3	500	1000	100	108	104	105	105	106	104	**103**
N4W3B3R7	500	1000	74	74	74	74	74	74	74	74
N4W4B1R0	500	1000	56	58	57	57	57	58	57	57
N4W4B1R1	500	1000	56	58	57	58	58	58	57	57

There are 10 large class instances that contain 200 items. The weights of these items are between 20,000 and 35,0000 and the bin capacity is 100,000. Our algorithm is executed on each instance 30 times and the best bin value, the average bin value with its standard deviation is given in Table 5. The same bin value is obtained in all of the runs for 4 instances, whereas the best bin value is obtained only once for HARD4 instance. On the other hand, the best bin value is obtained in more than half of the runs in the remaining 5 instances.

Table 6 shows the performance comparison of PBEA with the algorithms from the literature using the hard class instances. The best result reported in Table 5 is used as the performance of PBEA. Our algorithm outperforms the listed algorithms for 8 instances and gives the same performance with ACS and ILWOA in 2 instances. Even if we are very close to the best-known bin value for hard instances, unfortunately PBEA was not able to produce the best-known result.

Finally, the performance of the operators proposed in our algorithm is evaluated. We have conducted tests on the hard class instances and reported the best bin value with the average bin value and standard deviation of the results obtained in 30 runs in Table 7. There are two versions of the pool-based crossover operator which vary in the selection of the items in the pool to be placed to the bins. First crossover operator sorts all items in the pool from highest to lowest weight, whereas second operator shuffles the items in the pool. The first crossover type uses a sorted list, so it first tries to place the highest weight items to the bins and then tries the other items. Generally, it outperforms the second crossover type because placing heavier items to the bins first is a smarter decision since there will always be much space for lighter items as compared to heavier items. This sorting operation decreases the search space so when combined with local search, it shows worse performance as compared to shuffling. The pool-based crossover operator using shuffle may not be the best choice when considered alone, but with a combination of the local search technique, it gives the best results.

Table 5. Performance of PBEA on hard class dataset.

Instance	N	Capacity	BEST	AVG	STD
HARD0	200	100,000	57	57.37	0.49
HARD1	200	100,000	58	58.04	0.18
HARD2	200	100,000	58	58.37	0.49
HARD3	200	100,000	57	57	0
HARD4	200	100,000	58	58.97	0.18
HARD5	200	100,000	58	58	0
HARD6	200	100,000	58	58.23	0.43
HARD7	200	100,000	57	57	0
HARD8	200	100,000	58	58.2	0.41
HARD9	200	100,000	58	58	0

Table 6. Performance comparison of PBEA with related studies on hard class dataset.

Instance	N	Capacity	Best known	QICS	ACS	FCO	FA	AS	ILWOA	PBEA
HARD0	200	100,000	56	59	58	59	60	59	58	**57**
HARD1	200	100,000	57	60	59	59	60	60	59	**58**
HARD2	200	100,000	56	60	59	59	61	60	59	**58**
HARD3	200	100,000	55	59	58	59	60	59	58	**57**
HARD4	200	100,000	57	60	59	60	61	60	59	**58**
HARD5	200	100,000	56	59	58	59	60	59	58	58
HARD6	200	100,000	57	59	59	59	61	60	59	**58**
HARD7	200	100,000	55	59	58	58	59	59	57	57
HARD8	200	100,000	57	59	59	59	61	60	59	**58**
HARD9	200	100,000	56	59	59	59	60	59	59	**58**

Table 7. Performance of the operators of PBEA on hard class dataset.

	Crossover Only						Crossover + Local Search					
	Sorted			Shuffle			Sorted			Shuffle		
Instance	BEST	AVG	STD	BEST	AVG	STD	BEST	AVG	STD	BEST	AVG	STD
HARD0	59	59.1	0.31	59	59.04	0.18	58	58.07	0.25	57	57.37	0.49
HARD1	60	60	0	60	60	0	58	58.97	0.18	58	58.04	0.18
HARD2	60	60	0	60	60.04	0.18	59	59.23	0.43	58	58.37	0.49
HARD3	59	59	0	59	59	0	57	58.13	0.43	57	57	0
HARD4	60	60	0	60	60.5	0.51	59	59.8	0.41	58	58.97	0.18
HARD5	59	59.17	0.38	59	59.9	0.31	58	58.93	0.25	58	58	0
HARD6	59	59.94	0.25	60	60.1	0.31	59	59	0	58	58.23	0.43
HARD7	58	58.4	0.5	58	58.1	0.31	57	57.17	0.38	57	57	0
HARD8	60	60	0	60	60.14	0.35	59	59	0	58	58.2	0.41
HARD9	59	59.9	0.31	59	59.97	0.18	58	58.67	0.48	58	58	0

V. CONCLUSION AND FUTURE WORK

In this study, we have proposed a novel pool-based evolutionary algorithm that solves the one-dimensional bin packing problem. We have designed the pool-based crossover operator to increase the diversity and the combine and remap local search technique to increase the quality of the solution. The experimental study indicates that our algorithm outperforms six algorithms from the literature with respect to the total number of bins used.

The proposed algorithm can be used for the solution of real world problems such as industrial and logistic applications, multiprocessor scheduling and cloud computing that can be modeled using bin packing problem. Pool-based evolutionary algorithm can also be applied to two-dimensional or three-dimensional bin packing problem if the representation of the individuals is changed properly.

REFERENCES

[1] Garey, M., & Johnson, D. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W.H. Freeman Co..

[2] Johnson, DS., Demers, A., Ullman, JD., Garey, MR., & Graham, RL. (1974). Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4), 299–325.

[3] Rhee, WT., & Talagrand, M. (1993). On line bin packing with items of random size. *Mathematics of Operations Research*, 18(2), 438–445.

[4] Sensarma, D., & Sarma, SS. (2014). A novel graph based algorithm for one dimensional bin packing problem. *Journal of Global Research in Computer Science*, 5(8), 1–4.

[5] Layeb, A., & Benayad, Z. (2014). A novel firefly algorithm based ant colony optimization for solving combinatorial optimization problems. *International Journal of Computer Science and Applications*, 2(11), 19–37.

[6] Dorigo, M., Maniezzo, V., & Colorni, A. (1996) Ant system: optimization by a colony of cooperating agents. *IEEE Trans Syst Man Cybern Part B (Cybernetics)*, 26(1), 29–41.

[7] Zendaoui, Z., & Layeb, A. (2016). Adaptive Cuckoo Search Algorithm for the Bin Packing Problem. *Modelling and Implementation of Complex Systems*, 2, 107–120.

[8] Layeb, A., & Boussalia, S.R. (2012). A Novel Quantum Inspired Cuckoo Search Algorithm for Bin Packing Problem. *Information Technology and Computer Science*, 2(5), 58–67.

[9] Yang, X.-S. (2009). Firefly Algorithm, L´evy Flights and Global Optimization. *Research and Development in Intelligent Systems*, 2(26), 209–218.

[10] Abdel-Basset, M., Manogaran, G., Abdel-Fatah, L., & Mirjalili, S. (2018). An improved nature inspired meta-heuristic algorithm for 1-D bin packing problems. *Personal and Ubiquitous Computing*, 2(22), 1117-1132.

[11] Falkenauer, E. (1996). A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics*, 2(2), 5–30.

[12] Quiroz-Castellanos, M., Cruz-Reyes, L., Torres-Jimenez, J., G´omez, S. C., Fraire Huacuja, H., & Alvim, A. C. F. (2015). A grouping genetic algorithm with controlled gene transmission for the bin packing problem. *Computers and Operations Research*, 2(55), 52–64.

[13] Kirkpatrick, S., Gelatt, D., & Vecchi, M. P. (1983). Optimization by Simulated Annealing. *Science* 2(220), 671–680.

[14] Gen, M., & Cheng, R. (2000). *Genetic Algorithms and Engineering Optimization*. John Wiley Sons.

[15] Sungu, G., & Boz, B. (2015). An evolutionary algorithm for weighted graph coloring problem. In: Proceedings of the Companion In 2015 *Annual Conference on Genetic and Evolutionary Computation (GECCO)*, (pp. 1233–1236). ACM.

[16] Boz, B., & Sungu, G. (2020). Integrated crossover based evolutionary algorithm for coloring vertex weighted graphs. *IEEE Acess*, 8, 126743 – 126759.

[17] Scholl, A., Klein, R., & Jurgens, C. (1997). BISON: a fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Computers and Operations Research*, 2(24), 627–645.