Central European Journal of Operations Research (2021) 29:425–445
https://doi.org/10.1007/s10100-020-00695-5

A hybrid evolutionary algorithm for the offline Bin Packing Problem

Istvan Borgulya[1]

Received: 27 September 2019 / Accepted: 27 June 2020 / Published online: 15 July 2020

Abstract
In this paper we present an evolutionary heuristic for the offline one-dimensional Bin Packing Problem. In this problem we have to pack a set of items into bins of the same capacity, and the objective is to minimize the number of bins used. Our algorithm is a hybrid evolutionary algorithm where an individual is a feasible solution, and it contains the description of the bins. The algorithm works without recombination; it uses two new mutation operators and improves the quality of the solutions with local search procedures. The mutation operators' work is based on a relative pair frequency matrix, and, based on this matrix, we know the frequency of every pair of items i.e. how often they are included in the same bin in the best solutions. The frequency matrix helps to pack items into subsets of items; these subsets are the bins in our problem. The algorithm was tested on well-known benchmark instances from the literature and was compared with both evolutionary and state-of-the-art algorithms. Our algorithm achieved a valuable result with the difficult *hard28* test set, and in most of the test problems it reached the optimum.

Keywords Local search · Bin packing · Evolutionary algorithm

1 Introduction

In the offline one-dimensional Bin Packing Problem (BPP) we have a set of n items and an unlimited number of bins. Every bin's capacity is $c>0$ and every item i ($i=1$, 2, …, n) has a size $s_i > 0$ and $s_i \le c$. The goal is to pack the items into the minimum number of bins with the capacity of the bins not exceeded: for every bin k

$$\sum_{i \in bin(k)} s_i \le c.$$

✉ Istvan Borgulya
borgulya.istvan@ktk.pte.hu

1 University of Pecs Hungary, Rakoczi ut 80, 7621 Pecs, Hungary

BPP belongs to the ‘cutting & packing’ problems. There are many industrial and logistic applications of BPP such as loading trucks with weight limitations, stock-cutting problems where the bins correspond to standard lengths of some material (cable, paper) from which items must be cut (Coffman et al. 1996).

BPP is NP-hard (Garey et al. 1979), and many exact, heuristic and meta-heuristic algorithms have been published to solve it. It was, in fact, solved exactly using dynamic programming, LP relaxation, branch-and-bound, branch-and-price and constraint programming methods (see e.g. Delorme et al. 2016). The exact methods can give the optimal solution, and we can use them to solve small BPP cases in a reasonable time.

The approximation algorithms, heuristics and meta-heuristics are not guaranteed to find the optimal solution, but their running time is short. The first group of methods comprise the approximation algorithms which had their performances mathematically analysed. The most successful are the First-Fit Decreasing and Best-Fit Decreasing methods.

The heuristic methods are search algorithms that are able to find the global optimum only with a high degree of probability. They are usually repeated and so produce best and average results. The common heuristic methods are improved versions of the approximation algorithms, versions with local searches and heuristics generated with genetic programming (Burke et al. 2006). There are meta-heuristics for BPP also. We can find simulated annealing, tabu search, variable neighbourhood search and weight annealing (e.g. Alvim et al. 2004; Buljubašić and Vasquez 2016; Fleszar and Hindi 2002; Loh et al. 2008). Some methods use evolutionary techniques: ant colony optimization, evolutionary strategy and genetic algorithms (e.g. Bugger et al. 2004; Falkenauer 1996; Quiroz-Castellanos et al. 2015; Stawowy 2008). There are also hyper-heuristics combined with simulated annealing or genetic algorithms (e.g. Jiang et al. 2011; López-Camacho et al. 2011) and parallel grouping genetic algorithms also (e.g. Dokeroglu and Cosar 2014; Kucukyilmaz and Kiziloz 2018). A possible evolutionary method for BPP is the estimation of distribution algorithm (EDA) also (regarding EDA see e.g. Pelikan et al. 1999). The EDA estimates a probability distribution from a set of solutions and usually updates the estimated distribution in every generation. The new solutions are generated using the probability distribution (this is the sampling). For 3D BPP with various bin sizes we found a method, which applies an EDA (Cai et al. 2013). For one-dimensional BPP no similar EDA exists.

In this paper, our motivation was to build an evolutionary algorithm (EA) for BPP which gives a better result than the earlier evolutionary techniques and solves the difficult *hard28* test set successfully. For this we planned to use a relative pair frequency matrix. From this matrix (*RPFM*) we know the frequency of every pair of items also how often they are members of the same bin in the best solutions. On this basis we can pack items into subsets of items; these subsets are the bins in our problem. Higher values of frequency mean better pairs of items in the same bin. The *RPFM* is a modified version of the ECM matrix of the knapsack problems (see Borgulya 2019). (See *RPFM* in Sect. 2.)

For permutation problems we can also find models where there is a matrix based on variable pairs. (These include the Travelling salesman, Flow shop, Linear ordering and Quadratic assignment problems.) There is also a model based on consecutive variable pairs for permutation-based EDA (Cerebio et al. 2012). *The edge histogram model* estimates a probabilistic model that learns the adjacency of variables in the selected individuals at each generation (Tsutsui et al. 2003). The matrix of this model gives at every variable the probability of the nearest neighborhood variables in a permutation. The sampling generates a permutation based on the matrix. We use *RPFM* in another way. With our *RPFM* model we can generate subsets of items searching for appropriate pairs of items, and the permutation of the items is not important in a subset.

The basic features of our EA were:

- The individual contains a feasible solution—the bins with their items.
- In every generation we select one individual for descendent.
- In the individual we select fully filled and almost full bins into a special set (FB).
- We use two new mutation operators based on a relative pair frequency matrix. The mutation operators work only on the not full bins of the descendent; the FB bins are not modified.
- We use local searches to improve the results and modify the bins in FB.
- The algorithm is terminated if the running time limit is reached. The best solution will be the result.

In planning our EA we used first the mutation operators without local search procedures. Based on the test results, our EA could solve the not difficult instances optimally and, with the optimal number of bins plus one extra bin, the other instances. To improve the result, we next used local search procedures also. The EA applied repeatedly a group of local searches, 8–10 such searches consecutively (see local searches in Sect. 3.2). In most problems we reached the optimum. Our goal, however, was to solve the difficult *hard28* test set optimally also. Using local searches our algorithm solved all instances of *hard28* optimally without the extra bin.

Our contribution, therefore, is a new hybrid EA for BPP (named HEA) and its key features are:

- We use a relative pair frequency matrix to select items into subsets.
- Based on this relative pair frequency matrix, we can construct bins and solutions.
- We use two new mutation operators based on this matrix.
- In the individual we isolate the fully filled and almost full bins into a special set, these bins not being modified by mutations.

The remainder of our paper is organized as follows: Sect. 2 describes some important elements of the algorithm; Sect. 3 gives the main steps of the hybrid HEA algorithm. The computational results are reported in Sect. 4, and the conclusions form Sect. 5.

2 Preliminaries

Let us first examine some important elements of our algorithm. They are the fullness of a bin, the structure of the individual, the fitness function, the *RPFM* model, the bin-packing procedure, the initial population and the Unified Computational Time.

2.1 Fullness of a bin

For the operation of the HEA, we defined a *fullness proportion limit* (*fpl*). The values of *fpl* can be 0.99, 0.999 or 0.9999. If the fullness proportion of a bin from the individual is higher than *fpl* we say that the bin is full and belongs to the set of full bins (FB). The other bins of the individual belong to the group of not full bins (NFB).

HEA applies two mutation operators based on the *RPFM* (see mutation in Sect. 3). Both mutation operators work only on NFB of the descendent, and so the use of the FB set can improve the speed of the algorithm. If the FB set is empty, the running time of a generation is longer. Only local searches can modify the FB bins in every generation. These local searches keep these bins in FB; the bins remain fully filled bins and the fullness proportions of the bins do not decrease.

2.2 The structure of the individual

Every individual of the population contains the description of the bins. The individual contains all the important data: the number of bins, the number of items in each bin, the identification numbers of the items and the fullness proportion of the bin. If the fullness proportion of a bin is higher than *fpl* the bin belongs to the FB set—otherwise to the NFB set. The size of the individuals can be different.

2.3 Fitness function

The fitness function gives the quality of the packing that is given in the individual. It is used for comparing the packaging quality of individuals. We chose a fitness function based on (Burke et al. 2006; Falkenauer 1996). Our fitness function is

$$f = nb - \sum_{k=1}^{nb} (F_k/c)^2$$

where nb is the number of bins, F_k is the sum of the sizes of the items packed into the bin k ($k = 1,2, \ldots, nb$) and c is the capacity of a bin.

2.4 The *RPFM* model

With the relative pair frequency matrix we can estimate the probability that the *i*th and the *z*th items used to be in the same bin. Based on *RPFM* we can group items into subsets; these subsets are the bins in our problem. If there are items in a bin, we can search other items into the bin. We can accept other items if the estimated probabilities of all *i*th and *j*th items—where the *i*th is an already stored item in the bin and the *j*th is another item—are satisfactorily high.

Our starting point is the ECM matrix and technique from Borgulya (2019). Recently we have modified this technique in the following way: we have to know the frequency of every pair of items—how often they are in the same bin in the best individuals. As "best individuals" we take the best 20% of the population based on fitness values.

Let *RPFM* be an $n \times n$ matrix that stores the relative frequencies of the different pairs. Every item has a row and a column in the matrix. *RPFM* is a symmetric matrix since, if the *i*th and the *j*th items are a pair in a bin, then the *j*th and the *i*th items are also a pair in the same bin. In every pair the items are different, and so the values of the main diagonal in the matrix are null. The upper or lower triangular matrixes of *RPFM* give the frequency values. Using, for example, the upper triangular matrix without the main diagonal, we can reduce the elements of *RPFM*: there are $n*(n-1)/2$. In this case we can store the triangular matrix in a vector. The sizes of the problems are limited in our program by the possible size of a vector in our computer and the maximum size of this vector depends on the other declarations too (our program could manage problems with a maximum $n=18{,}000$ items. If n is higher than the possible maximum value of n, the program gives an error message). We will use the upper triangular matrix of *RPFM* in the description.

RPFM is updated throughout the evolution process using the "best individuals". We update *RPFM* after every *kgen*th generation (e.g., $kgen=10$). The updating process is as follows:

Let *ΔRPFM* be a similar triangular matrix to *RPFM*. It will be a working matrix during the update. Let $RPFM_{ij}$ be the collected relative frequency of the *i*th and the *j*th item (a pair) in common bins until a given *gen*th generation. We can update the elements of the *RPFM* matrix with the element of *ΔRPFM*

$$RPFM_{ij} = (1-\alpha) * RPFM_{ij} + \alpha * \Delta RPFM_{ij}$$

where $\Delta RPFM_{ij}$ is the relative frequency of the *i*th and the *j*th items in common bins based on the "best individuals" of the *gen*th generation and α denotes some relaxation factors (e.g., $\alpha=0.2$). *Algorithm* 1 gives this process and Fig. 1 gives an example of the update of *RPFM*.

Algorithm 1. Update-RPFM procedure

```
1.  Every value of ΔRPFM is 0.
2.  We take the best 20% of the population based on the fitness values. They will be the "best individuals".
3.  for (i=1,2,...,n-1)
        for (j=i+1,...,n)
            We count how many times the ith and jth items are in common bins in the "best individuals". We take
            into account all bins from the "best individuals" where the number of items >1.
        end
    end
4.  We divide the ΔRPFM matrix by the number of these "best individuals".
5.  We update RPFM with ΔRPFM:
    for (i=1,2,...,n-1)
        for (j=i+1,...,n) $RPFM_{ij} = (1-\alpha) * RPFM_{ij} + \alpha * \Delta RPFM_{ij}$
    end
```

We use the *RPFM* matrix to estimate the probability of pairs of items. The formula

Consider a BPP instance with bin capacity equal to 10 and 9 items {1, ... ,9} with sizes (6, 3, 7, 8, 5, 2, 2, 5, 2). Let the size of the population be 10. Every value of *ΔRPFM* and *RPFM* are 0. At the first update of *RPFM* let the two best individuals be from the population as follows:

Bin 1 is packing items 2 and 3 and its fullness proportion is 1

bin	1	2	3	4	5
items	2,3	4	1,9	5,7	8,6
fullness prop.	1	0.8	0.8	0.7	0.7

bin	1	2	3	4	5
items	4,6	2,8	9,3	5,7	1
fullness prop.	1	0.8	0.9	0.7	0.6

After the Update-RPFM procedure the *ΔRPFM* and *RPFM* are the following:

$$\Delta RPFM = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.5 \\ & 0 & 0.5 & 0 & 0 & 0 & 0 & 0.5 & 0 \\ & & 0 & 0 & 0 & 0 & 0 & 0 & 0.5 \\ & & & 0 & 0 & 0.5 & 0 & 0 & 0 \\ & & & & 0 & 0 & 1 & 0 & 0 \\ & & & & & 0 & 0 & 0.5 & 0 \\ & & & & & & 0 & 0 & 0 \\ & & & & & & & 0 & 0 \\ 0 & & & & & & & & 0 \end{bmatrix} \quad RPFM = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.1 \\ & 0 & 0.1 & 0 & 0 & 0 & 0 & 0.1 & 0 \\ & & 0 & 0 & 0 & 0 & 0 & 0 & 0.1 \\ & & & 0 & 0 & 0.1 & 0 & 0 & 0 \\ & & & & 0 & 0 & 0.2 & 0 & 0 \\ & & & & & 0 & 0 & 0.1 & 0 \\ & & & & & & 0 & 0 & 0 \\ & & & & & & & 0 & 0 \\ 0 & & & & & & & & 0 \end{bmatrix}$$

At the second update of *RPFM* let the two best individuals be from the population the following:

bin	1	2	3	4	5
items	4,6	2,7	9,3	1,7	4
fullness prop.	1	0.8	0.9	0.8	0.8

bin	1	2	3	4	5
items	4,6	9,3	1,7	2,8	5
fullness prop.	1	0.9	0.8	0.8	0.8

After the Update-RPFM procedure the *ΔRPFM* and *RPFM* are the following:

$$\Delta RPFM = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 0 & 0.5 & 0.5 & 0 \\ & & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ & & & 0 & 0 & 1 & 0 & 0 & 0 \\ & & & & 0 & 0 & 0 & 0 & 0 \\ & & & & & 0 & 0 & 0 & 0 \\ & & & & & & 0 & 0 & 0 \\ & & & & & & & 0 & 0 \\ 0 & & & & & & & & 0 \end{bmatrix} \quad RPFM = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0.2 & 0 & 0.08 \\ & 0 & 0.08 & 0 & 0 & 0 & 0.1 & 0.18 & 0 \\ & & 0 & 0 & 0 & 0 & 0 & 0 & 0.28 \\ & & & 0 & 0 & 0.28 & 0 & 0 & 0 \\ & & & & 0 & 0 & 0.16 & 0 & 0 \\ & & & & & 0 & 0 & 0.08 & 0 \\ & & & & & & 0 & 0 & 0 \\ & & & & & & & 0 & 0 \\ 0 & & & & & & & & 0 \end{bmatrix}$$

Fig. 1 Example of the update of *RPFM*

$$pr_{ij} = RPFM_{ij} / \sum_{t=1}^{n} RPFM_{it}$$

gives the estimated probability that the *i*th and the *j*th items are in the same bin. If pr_{ij} is high the two items will be probably together in a bin.

2.5 Bin-packing procedure

Our algorithm is a bin-oriented heuristic which constructs solutions by packing one bin at a time (see Fleszar 2002). It constructs an initial solution or constructs bins from subsets of items. Let Q be the working set of the bin-packing procedure and the items for packing are in Q. *Bin-packing* constructs from the items of Q new bins using a similar technique to the sampling technique of an EDA.

Bin-packing selects items for the bins and constructs one bin at a time. For every bin it selects a new set of items from the not yet selected items of Q. Algorithm 2 gives this process and Fig. 2 gives an example of Bin-packing.

Algorithm 2. Bin-packing procedure

```
1. Let the items be in Q for packing.
2. Filling a bin
    a. Initializing the bin
       If there are unselected items in Q, first it chooses a random item $x$ in the latter case; if there are
       none, go to step 4. The item $x$ will be the first element in the bin.
    b. Filling up the bin
       Next, we search pairs matching the item $x$ based on the RPFM. Looping over the items $j$ that are
       not yet in a bin, the process includes them in the bin – if at every already stored $k$th item from
       the bin
           $rnd < pr_{kj}$ (the $k$th and $j$th items are a pair)
       where $rnd$ is a random number from [0,1] and the total size of the previous items of the bin plus
       the size of the item $j$ do not violate the capacity of the bin.
    c. Finalizing the bin
       Two local search procedures improve the fullness of the bin. The repair1 procedure searches
       unselected items with the largest size from Q that it can insert into the bin without violating the
       capacity. The repair2 procedure searches unselected items from Q and it can swap every
       unselected item with one item from the bin if it improves the fullness of the bin without
       violating the capacity. Both procedures are repeated until there is no further improvement.
3. If there are unselected items in Q, it repeats the process with a new bin (go to step 2).
4. The bins are ready.
```

2.6 Initial population

We generate the individuals of the initial population with the *Bin-packing* procedure. In the first steps of the algorithm the elements of the matrix are 0, and so the *Bin-packing* procedure is not able to use the *RPFM* matrix to select items. With every bin it chooses an item at random (this is the first element in the bin) and, after *repair1* and *repair2* procedures, can fill the bin. The resulted bins are random, feasible individuals as the initial population.

Consider again the BPP instance with bin capacity equal to 10 and 9 items {1, …, 9} with sizes (6, 3, 7, 8, 5, 2, 2, 5, 2). Let Q be {2, 3, 4, 5, 6} and *RPFM* is the last matrix from the earlier example. The steps of *Bin-paking* are the following:

- It chooses an item at random for bin 1: *4*th item.
- It searches for appropriate (4, j) item pairs; supposed that there are none.
- It applies the *repair1* procedure: it can insert the *6*th item. The bin is full; it does not apply *repair2* and bin 1 is ready. In bin 1 the items are *4*th and *6*th.
- It chooses an item at random for bin 2: *2nd* item.
- It searches for (2, j) item pairs; assume that the (2,3) item pair is good. The bin is full and so bin 2 is ready. Then in bin 2 the items are *2nd* and *3rd*.
- It chooses an item at random for bin 3: *5*th item.
- The bins are ready.

Fig. 2 Example of *Bin-packing*

2.7 Unified computational time

The methods of the comparative results section were executed on different machines, and so ″we calculated appropriate scaling factors to compare their running times. For this purpose, we used the CPU speed estimations provided in the SPEC standard benchmark″ (https://www.spec.org/cpu2006/results/cint2006.html) (Buljubašić and Vasquez 2016). Based on the SPEC standard, we obtained CPU speeds for the different processors. With the CPU speeds we can calculate appropriate scaling factors to compare the running times of the different computers. We chose the CPU speed of the computer of a method as a reference, and the scaling factors are at a CPU: CPU speed/ reference CPU speed. Multiplying the CPU time of a processor by its scaling factor, we obtain a Unified Computational Time (UCT) for comparing their running times. (See Buljubašić and Vasquez 2016; Quiroz-Castellanos et al. 2015).

3 The HEA algorithm

Our HEA generates only one descendent in every generation. First comes the initial population based on *RPFM*, and next, in every generation it selects an individual for descendent, applies a mutation operator and improves the result with local searches (LS).

For certain tasks, the algorithm might be ‘’stuck’’ at one of the local optima. To enable escape towards a potential global optimum, the algorithm generates new, additional individuals. A new individual is also a descendent and can help to improve the capability and the speed of the algorithm to find the global optimum. Thus, new descendants are periodically inserted in the population until the maximum size of the population is reached.

Algorithm 3 shows the main steps of HEA. The parameters of the algorithm are as follows:
tmax – the maximal size of the population.
tin – the first size of the population.
kgen – the algorithm is controlled in every *kgen*th generation.
timeend – the limit of the running time.
fpl – the fullness proportion limit.
The next parameters will be explained later in this section:
LSn, LSm, p_{ls} – parameters of the local searches.

```
Algorithm 3. The main steps of HEA
    Input: the instance, the values of the parameters.
    Output: best_solution.
    Every value of RPFM is 0.
    t=tin. // t – the size of the population during the execution.
    Construct the initial population with Bin-packing.
    Update-RPFM.
    Repeat
        Do kgen times
            Apply selection.
            Apply mutation.
            Apply local searches. Reinsertion.
        od
        If (t<tmax) then t=t+1 fi
        Apply local searches on the best individual.
        Reinsertion. Update-RPFM.
    until running time>timeend
end
```

The operation of HEA is as follows:

Input The algorithm reads the instance and the values of the parameters (described in Sects. 4.1 and 4.2). Every item has a unique identification number.
Selection. The algorithm selects an individual based on truncation selection. In this, only the best *tp* percentage of the population are considered as a potential parent.
Mutation The algorithm applies the mutation based on the p_m parameter. If p_m =0, it applies *mutation1;* if p_m =1, it applies *mutation2* and if p_m =0.5, it applies *mutation1* with probability 0.5 and also with probability 0.5 applies *mutation2* (see Sect. 3.1).
Local searches The algorithm applies local searches *LSn* times on the descendent, and *LSm* times on the best individual. The local searches improve the fullness of the bins, or increase the diversity of the population (see Sect. 3.2).
Reinsertion This is a crowding technique that compares the descendent with the parent. The descendent may replace the parent if the descendent is better. If the

descendent is an additional individual, the new descendent is inserted without any further analysis into the population.

Stopping criterion The algorithm is terminated if the running time limit is reached.

3.1 Mutation

There are two mutation operators: *mutation1* and *mutation2*.

Mutation1 The *mutation1* is a swap of items between two NFB bins. The first bin has to have more than two items; otherwise it does not consider the bin. (If there is no appropriate bin, *mutation1* is finished). The mutation is based on the *RPFM*, and so it selects the *i*th and the *j*th items from the given bin with the largest pr_{ij} and selects at random another *k*th item from the bin. If the *i*th and *k*th items are in the same bin of the best individual, *mutation1* is finished. Otherwise it chooses the *z*th item with the largest pr_{iz} probability from a different bin, if the swap does not violate the capacity of the bins and swaps the *k*th *z*th items between the bins.

It repeats *mutation1* on the descendent *it* times, where *it* is a random integer from [1, n/2].

Mutation2 The second mutation operator uses the *Bin-packing* procedure to generate new bins in the descendent. First it deletes the NFB bins and stores their items in a Q set. Next it constructs bins from the items of Q with the *Bin-packing* procedure. If there are bins in FB they will be elements of the descendent also.

Figure 3 shows examples of *mutation1* and *mutation2*

3.2 Local searches

The algorithm improves the descendent with a group of LSs. It applies the LSs from the group one after the other (e.g. *LS1*+ *LS2* + *LS3*) and repeats the group *LSn* or *LSm* times. It repeats the group *LSn* times at every descendent, and *LSm* times at the best solution. It applies every local search from the group with p_{ls} probability.

Usually the FB bins do not influence the search; the algorithm can find the optimal solution based only on the items of NFB. We may arrive at an NFB that does not permit finding the optimal solution, and so we allow the swapping of items among the FB and NFB bins with the help of local search procedures. These local searches keep these bins in FB; they remain fully filled bins and the fullness proportion of the bins does not descend.

There are 10 local search procedures: *LS1, LS2, ..., LS10*. *LS1, LS2, LS3, LS4* and *LS5* swap the items between FB and NFB, whilst the other *LS6, ..., LS10* work on NFB and try to improve the solution with various moves. The algorithm applies at every descendent *LSn* time the following group: *LS1, LS2, LS4, ..., LS10*; at the best individual *LSm* times it applies the following group: *LS1, LS2, LS3, LS4, LS7, LS9.*

Consider again the BPP instance with bin capacity equal to 10 and 9 items {1, … ,9} its sizes (6, 3, 7, 8, 5, 2, 2, 5, 2). Let a parent and *RPFM* be the following:

Parent:

bin	1	2	3	4	5
items	4,6	2,8	3	5,7,9	1
fullness prop.	1	0.8	0.7	0.9	0.6

$$RPFM = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0.2 & 0 & 0.08 \\ & 0 & 0.08 & 0 & 0 & 0 & 0.1 & 0.18 & 0 \\ & & 0 & 0 & 0 & 0 & 0 & 0 & 0.28 \\ & & & 0 & 0 & 0.28 & 0 & 0 & 0 \\ & & & & 0 & 0 & 0.16 & 0 & 0 \\ & & & & & 0 & 0 & 0.08 & 0 \\ & & & & & & 0 & 0 & 0 \\ & & & & & & & 0 & 0 \\ 0 & & & & & & & & 0 \end{bmatrix}$$

Mutation1: for *mutation1* only bin 4 is appropriate (with more than two items). In bin 4 the item pairs are (5,7), (5,9) and (7,9). (5,9) has the largest pr_{ij}, and so the selected item will be 7. Suppose that the (5,7) pair is not in the same bin of the best individual; then we can swap item 7 only for item 2. The descendent is as follows:

bin	1	2	3	4	5
items	4,6	7,8	3	5,2,9	1
fullness prop.	1	0.7	0.7	1	0.6

(We repeat *mutation1* only one time.)

Mutation2: using the same parent and *RPFM,* we first select the full bins (bin 1). From the other bins we store the items in Q {2, 8, 3, 5, 7, 9, 1} and delete these bins. Suppose that the *Bin-packing* procedure generates only 3 bins from the items of Q; the descendent is as follows:

bin	1	2	3	4
items	4,6	2,3	1,7,9	5,8
fullness prop.	1	1	1	1

Fig. 3 Examples of mutation

Local searches between FB and NFB.

- *LS1 procedure* This tries to swap every item pair from a random bin of FB with 1 to 3 NFB items. First it copies all items from the NFB bins to a vector VITEM. Between all possible item pairs from the FB bin and items from VITEM the procedure searches for an appropriate swap where the FB bin remains fully filled. At the end the procedure constructs a new NFB from the items of VITEM based on *RPFM*.
- *LS2 procedure* This tries to swap every item pair from one of all possible FB bins with 1 item from an NFB bin. The procedure searches for appropriate swaps where the fullness proportions of the FB bins do not descend.

- *LS3 procedure* This tries to swap every item pair from one of all possible FB bins with 2 items from an NFB bin. The procedure searches for appropriate swaps where the fullness proportions of the FB bins do not descend.
- *LS4 procedure* This tries to swap every item from one of all possible FB bins with 2 items from an NFB bin. The procedure searches for appropriate swaps where the fullness proportions of the FB bins do not descend.
- *LS5 procedure* To improve the fullness of the bins it tries to insert one item to every FB bin where the fullness proportion < 1. It inserts the items from the NFB bins where the fullness proportion $< = 0.9$.

Local searches of NFB.

- *LS6 procedure* To improve the fullness of the bins it tries to insert 1–1 items to every NFB bin where the fullness proportion > 0.9. It inserts the items from the NFB bins where the fullness proportion $< = 0.9$.
- *LS7 procedure* This chooses all possible NFB bin pairs. For each pair of bins it creates a new bin pair by re-arranging their contents, aiming to reach a higher fullness than the maximum fullness of the two original bins. Creating the new bins, the procedure uses blocks from 1 to 3 items. If in the new bin pair there is no higher fullness, the procedure does not accept the new bin pair.
- *LS8 procedure* This chooses all neighbouring bin pairs from NFB and every pair it tries to swap with a new bin pair sampling *RPFM*. (If the process generates more than two bins, the algorithm does not accept the bins and skips the swap.)
- *LS9 procedure* This chooses a random NFB bin, tries to improve its fullness and tries to achieve smaller fitness. *LS9* performs swaps between the random bin and other bins. For this, the algorithm searches bins with smaller fullness than the random bin, builds pairs with the random bin and tries a 1–1 item swap between the bins. It accepts a swap if the fullness of the random bin is larger and the fitness is smaller after the swap.
- *LS10 procedure* For all possible bin pairs from NFB:
    - This tries to improve the fitness, swapping 1 item with 2 items between the bins of the pair. It accepts a swap if the fitness value is improved. Next.
    - It tries to improve the fitness, swapping 2 items with 2 items between the bins of the pair. It accepts a swap if the fitness value is improved. Next.
    - it tries to improve the fitness, swapping 1 item with 1 item between the bins of the pair. It accepts a swap if the fitness value is improved.

4 Experimental results

The HEA algorithm was implemented in C++. It was executed on iMAC with an Intel Core i5 2.5 GHz processor with 16 GB of RAM, running the MacOS Sierra 10.12.2 operating system.

We tested our algorithm on benchmark instances that are used in general in publications. The instance sets are available at http://or.dei.unibo.it/library/bpplib. The test sets are as follows: the *U* and *T* test sets of Falkenauer; three sets (*set-1, set-2, set-3*) of Scholl, Klein and Jürgens; the *gau* test set of Wäscher and Gau; two test sets (*was-1, was-2*) of Schwerin and Wäscher and the *hard28* test set of Schoenfield. The description of the problem instances is available in Table 1. The table gives the test sets (sets), the number of instances in a set (Num), the numbers of items in the instance (n), the capacity (cap), the range of item size (item size) and the range of optima.

4.1 Parameter selection

We analysed the process of HEA to determine how the parameter values affect convergence. From the 1615 test instances we chose 68 for parameter selection. They are the *U1000* instance groups of *U*, the *T60* instance group of *T* and the *hard28* data set.

Because our algorithm has a similar structure and parameters to our earlier algorithm in Borgulya (2019), we could accept the earlier parameter values. These parameters are the population size (*tin* and *tmax* parameters), the frequency of checks (*kgen* parameter), the generation in the first stage (*itt* parameter) and of the truncation selection (*tp*). The accepted parameter values are $tin = 5$, $tmax = 30$, $itt = 5$, $kgen = 5$ and $tp = 0.1$.

Table 1 Description of the problem instances

Sets		Num	n	cap	item size	Range of optima
U	*U120*	20	120	150	[20, 100]	[46, 52]
	U200	20	200			[99, 106]
	U500	20	500			[196, 207]
	U1000	20	1000			[393, 411]
T	*T60*	20	60	1000	[250, 500]	20
	T120	20	120			40
	T249	20	249			83
	T501	20	501			167
set-1		720	{50, 100, 200, 500}	{{100, 120, 150}}	[1, 100]	[15, 373]
set-2		480	{50, 100, 200, 500}	1000	[13, 627]	[6, 172]
set-3		10	200	100000	[20,000, 35,000]	[55, 57]
was-1		100	100	1000	{{150, 200}}	18
was-2		100	120	1000	{{150, 200}}	[21, 22]
gau		17	[57, 239]	10,000	[2, 7332]	[11, 28]
hard28		28	{160, 180, 200}	1000	[1, 800]	[58, 84]
total		1615				

For the time limit there are various values in the literature. Using our algorithm, we found that a duration of 60 CPU seconds is sufficient in 95% of the test problems. If $n > 150$ or the instances are harder, the problems can be time-consuming. In these cases, we use 300 CPU seconds. Hence the time limit is 60 or 300 s ($timeend = 60$ or $timeend = 300$).

The parameter values of FB, and of LSs are new parameters in HEA.

- *Fullness proportion limit* (*fpl*). We tested three values of *fpl*: 0.99, 0.999 and 0.9999. These values are appropriate for the instances.
- *LSn, LSm,* p_{ls} parameters of the local searches. p_{ls} is the probability of every local search. At every descendent p_{ls} can be 0.5 or 1, and at the best individual p_{ls} is 1. The values of the *LSn*, *LSm* and p_m parameters we analysed together, and Table 2 shows the results. At the selected test instances, we tried different combinations of the parameter-values. The table shows the number of instances where optimal solutions were found (*opt_f*). We can conclude the following:
  - without LS we can solve only a few instances optimally,
  - for *LSn, LSm* we found more appropriate values if p_m =0.5. The best results we achieved at the p_m =0.5, $LSn = 15$, $LSm = 30$ values, but we have good results at many instances with the values p_m =0.5, $LSn = 2$, $LSm = 2$ *also.*

Our goal was to give appropriate parameter values that will be good for every test set, for every instance. These parameters can be $tin = 5$, $tmax = 30$, $itt = 5$, $kgen = 5$, $tp = 0.1$, $fpl = 0.99$, p_{ls} =0.5 or 1, p_m $=0.5$, $LSn = 15$, $LSm = 30$ and $timeend = 60$ or $timeend = 300$.

4.2 Computation experience

HEA was run 10 times on each test instance of the test sets, and we provide the best results for every instance. Table 3 gives a summary of the results.

Table 3 shows the names of the test sets (*sets*), the number of instances in a test set (*inst*), the number of instances from the set where optimal solutions were found

Table 2 The results with different parameter values

Mutation, local search parameters	opt_f		
	U1000	T60	hard28
$p_m = 0, LSn = 0, LSm = 0$	2	0	5
$p_m = 1, LSn = 0, LSm = 0$	0	0	5
$p_m = 0.5, LSn = 0, LSm = 0$	2	0	5
$p_m = 0.5, LSn = 2, LSm = 0$	20	2	14
$p_m = 0.5, LSn = 0, LSm = 2$	20	8	14
$p_m = 0.5, LSn = 0, LSm = 30$	20	17	16
$p_m = 0.5, LSn = 2, LSm = 2$	20	14	18
$p_m = 0.5, LSn = 15, LSm = 15$	20	18	23
$p_m = 0.5, LSn = 15, LSm = 30$	20	20	28

(*opt_f*) and the average running time to the best solutions (*time*) (CPU time in seconds). We see in the table that HEA did not solve optimally only four instances. These instances are: N3C3W4_C, N4C3W4_S from set-1 and HARD2, HARD3 from set-3. In these cases the four instances were solved with plus 1–1 extra bin.

We now can give more detailed results of the *hard28* set. This is the most difficult test set based on the publications. In fact, most heuristics proposed for the BBP, including the best performers, cannot solve to optimality more than 5 instances from the *hard28* set (Buljubašić and Vasquez 2016).

In Table 4 we show the results of every instance. In the table we see the instance name (*instance*), n (*n*), the optimal values (or the lower bound) (*opt*), the number of optimal solutions found at an instance in 10 runs (*Hits*) and the running time to the best solution (*time*) (CPU time in seconds). HEA solved every instance of *hard28* optimally. If in a run HEA did not find the optimum, the result was larger only by one extra bin. (There were 173 extra bins in the 10 run.) Fig. 4 shows the convergence behaviour of HEA on *hard28*. We ran 6 instances 5–5 times. Figure 4 gives the average number of bins at 1, 5, 10, 15, 20, 40, 50, 85, 100, 105, 130 s during the execution time of HEA. We can see that HEA found in the case of each instance – and already in the first seconds – the optimum or optimal number of bins plus one extra bin.

4.3 Comparative results

(a) Comparing HEA to the best evolutionary algorithms.

For a comparison of the results we chose two EAs: the C-BP (Singh and Gupta 2007) and the CGA-CGT (Quiroz-Castellanos et al. 2015). C-BP is a combined method: a genetic algorithm and a perturbation MBS heuristic together. It was executed on a Pentium IV processor at 2.4 GHz with 512 MB RAM and was implemented in C. CGA-CGT is a genetic algorithm and was executed on an Intel Core2 Duo processor E6300 1.86 GHz. It was implemented in C++.

Table 3 The results for the test sets

Sets	Inst.	opt_f	Time
U	80	80	1.48
T	80	80	1.89
set-1	720	718	3.09
set-2	480	480	0.18
set-3	10	8	1.95
was-1	100	100	0.34
was-2	100	100	0.47
gau	17	17	0.39
hard28	28	28	19.02
Total	1615	1611	

Table 4 The results of HEA on the *hard28* set

Instance	n	opt	Hits	Time	Instance	n	opt	Hits	Time
BPP119	200	77	10	1.53	BPP531	200	83	1	72.10
BPP13	180	67	9	4.46	BPP561	200	72	9	0.67
BPP14	160	62	10	0.02	BPP60	160	63	6	10.94
BPP144	200	73	9	0.76	BPP640	180	74	2	53.05
BPP175	200	84	10	1.37	BPP645	160	58	10	2.15
BPP178	200	80	2	16.75	BPP709	180	67	9	1.55
BPP181	180	72	2	70.95	BPP716	180	76	10	0.02
BPP195	180	64	10	3.25	BPP742	160	64	2	180.37
BPP359	180	76	10	1.02	BPP766	160	62	5	73.16
BPP360	160	62	7	27.17	BPP781	200	71	10	0.91
BPP40	160	59	6	2.96	BPP785	180	68	7	1.78
BPP419	200	80	1	42.31	BPP814	200	81	1	14.27
BPP47	180	71	3	85.43	BPP832	160	60	8	6.58
BPP485	180	71	2	20.98	BPP900	200	75	2	23.90

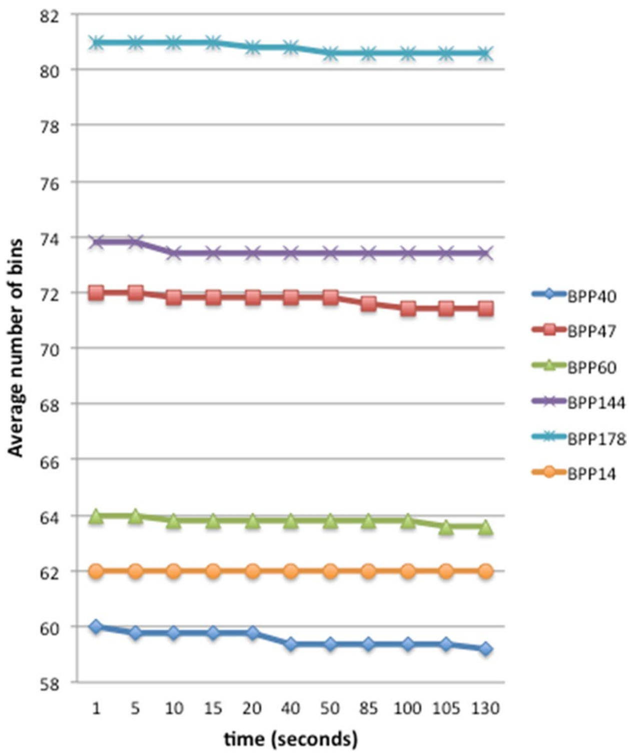

Fig. 4 Convergence behaviour of HEA on six instances of *hard28*

Table 5 Computer speed measures

Method	Processor	CPU speed	Scaling factor
P-SAWMBS	Intel core2 Q8200 2.33 GHz	18.3	1.48
CGA-CGT	Intel core2 duo CE6300 1.86 GHz	12.3	1
CNS-BP	Intel i7-3770 3.4 GHz	53.8	4.37
Belov	Intel Xeon 3.1 GHz	65.8	5.34
C-BP	Pentium IV 2.4 GHz	11.5	0.93
HEA	Intel core i5 2.5 GHz	30.5	2.47

Table 6 A comparison of the best evolutionary algorithms

Sets	Inst.	C-BP		CGA-CGT		HEA	
		opt_f	UCT	opt_f	UCT	opt_f	UCT
U	80	79	0.68	80	0.23	80	3.66
T	80	80	0.04	80	0.41	80	4.67
set-1	720	719	0.29	720	0.35	718	7.63
set-2	480	480	0.04	480	0.12	480	0.44
set-3	10	10	0.86	10	1.99	8	4.82
was-1	100	100	0.00	100	0.00	100	0.84
was-2	100	100	0.00	100	1.07	100	1.16
Gau	17	15	4.05	16	0.27	17	0.96
hard28	28	–	–	16	2.40	28	46.98
Total	1615	1584		1602		1611	

The methods of the comparative results section were executed on different machines, and so we calculated appropriate scaling factors to compare their UCT running times. We chose the CPU speed of the computer of CGA-CGT as a reference, and the scaling factors used are available in Table 5.

Table 6 shows the comparison. The table gives the name of the test sets (*sets*), the number of instances in a test set (*inst*), the number of instances from the set where optimal solutions were found *(opt_f)* and UCT times. We see the results of the three methods. They are very similar, and only with the *hard28* test set are larger differences found. HEA is the best with *hard28* and among the methods has the best result with the 1611 total number of optimal solutions.

If we examine the running times of the *U, T*, …, *gau* test sets, CGA-CGT is the fastest EA. Our HEA is the third and 5 times slower than CGA-CGT. However, at the *hard28* set we cannot compare times, as only HEA solved the hard instances successfully.

(b) Comparing HEA to the state-of-the-art methods.

For comparison of the results we chose three state-of-the-art methods: the *Belov* (from Delorme et al. 2016), the *P-SAWMBS* (Fleszar and Charalambous

2011) and *CNS_BP* (Buljubašić and Vasquez 2016). *Belov* is an exact branch-and–price method, the best method among 15 exact methods on the above problem sets (based on Delorme et al. 2016). It was executed on an Intel Xeon 3.10 GHz processor with 8Gbyte RAM and was implemented in C++. *P-SAWMBS* is a variable neighbourhood search variant and was executed on Intel PC Core2 at 2.33 GHz processor with 2 GB RAM. It was implemented in C. *CNS_BP* is a tabu search and was executed on an Intel Core i7-3770 CPU 3.40 GHz processor and was implemented in C++.

Table 7 shows the comparison. The first column gives the name of the test sets; the second shows the instance number within the test set. The *opt_f* column shows the number of instances from the set where optimal solutions were found by the methods, and the *UCT* column gives the comparable running times of the methods. Comparing the *opt_f* values we found larger differences only with the *hard28* test set. The other test sets were solved successfully or only with 1–2 mistakes. *P-SAWMBS* had a mistake in the *U* and the *gau* test sets and *HEA* had 2–2 on the *set-1* and *gau* sets. *Belov* had a time limit of 60 s and used only subsets of the instances at the U, set-1 and set-2 sets (the stars show). It solved the *T* sets with 23 mistakes (If the time limit were 10 min *Belov* successfully solved all the instances).

The *hard28* was the most difficult test set. *P-SAWMBS* could find the optimum only in 5 cases, *CNS_BP* in 25 cases. *HEA* and *Belov* solved the *hard28* set in all cases, and so *HEA* is the best heuristic method on the *hard28* set. Considering the total number of optimal solutions, we can conclude that the results of *CNS_BP* and *HEA* are similar; they have 3–4 mistakes and are the best heuristic methods.

Comparing running times, we have a different result. *P-SAWMBS* has the shortest running time with most of the test sets: on *U, T, set-1, set-2, was-1, was-2* and *gau*. On the *set-3* set *CNS_BP* is the fastest method and *P-SAWMBS* is the second fastest method. On these test sets the *HEA* is the third based on running time. At the *hard28* set we can compare only *Belov* and *HEA*, because only they solved

Table 7 A comparison with state-of-the-art methods

Sets	Inst.	Below		P-SAWMBS		CNS-BP		HEA	
		opt_f	UCT	opt_f	UCT	opt_f	UCT	opt_f	UCT
U	80 (74*)	74	0.00	79	0.00	80	0.30	80	3.66
T	80	57	131.90	80	0.00	80	0.09	80	4.67
set-1	720 (323*)	323	0.00	720	0.02	720	0.32	718	7.63
set-2	480 (244*)	244	1.60	480	0.00	480	0.14	480	0.44
set-3	10	10	75.30	10	0.24	10	0.08	8	4.82
was-1	100	100	5.34	100	0.00	100	0.00	100	0.84
was-2	100	100	7.48	100	0.02	100	0.00	100	1.16
gau	17	17	0.53	16	0.06	17	11.70	17	0.96
hard28	28	28	38.98	5	0.36	25	31.50	28	46.98
total	1615 (976*)	953		1590		1612		1611	

the hard instances successfully. Fleszar and Charalambous reported that their *P-SAWMBS* method could not solve more instances in the *hard28* set to optimality than the First Fit Decrease procedure (5 out of the 28), even after drastically increasing the maximum number of iterations in their algorithm (Buljubašić and Vasquez 2016). So, in this case *Belov* is the faster method.

Our goal was to build an EA for the problem, which gives better results than the earlier evolutionary techniques. The comparison between *CGA-CGT* and our algorithm shows that *CGA-CGT* is faster than our evolutionary algorithm, but in general *HEA* has better results, has fewer fault (Table 8 gives the execution times in UCT of the compared methods).

Can we reduce running times? Based on the parameter selection, we use the parameter values appropriate for solving both the easy and the hard problems also ($p_m = 0.5$, $LSn = 15$, $LSm = 30$). However, in Table 2 we can see that we can solve the easier problem with fewer repetitions of local searches. E.g. at U1000 we can solve the problems successfully with the p_m =0.5, $LSn = 15$, $LSm = 30$ and with the $p_m = 0.5$, $LSn = 2$, $LSm = 0$ parameter values also. The average running time to the best solution in the first case is 4.58 s and in the second case is 0.54 s. We obtained the results about 8 times faster in the second case. Hence, if we abandon the idea of common parameter values, with the easier problems we can reduce the number of repetitions of the local searches, and the average running times will be shorter. For this we should organize a hyperheuristic, where the appropriate values of the parameters will be searched for every problem.

5 Conclusion

In this paper we have presented a hybrid evolutionary algorithm for the BPP. Our algorithm, HEA, uses two new mutation operators and 10 local searches to improve the solutions. A relative pair frequency matrix helps to construct bins and the mutation operators and some of the local searches use this matrix also. Our test results are good. On the *hard28* test set our algorithm outperforms the earlier heuristics; it found the optimal solution for all instances. Based on our

Table 8 Execution times in UCT of the methods compared

	C-BP	CGA-CGT	HEA	Below	P-SAWMBS	CNS-BP
U	0.68	0.23	3.66	0.00	0.00	0.30
T	0.04	0.41	4.67	131.90	0.00	0.09
set-1	0.29	0.35	7.63	0.00	0.02	0.32
set-2	0.04	0.12	0.44	1.60	0.00	0.14
set-3	0.86	1.99	4.82	75.30	0.24	0.08
was-1	0.00	0.00	0.84	5.34	0.00	0.00
was-2	0.00	1.07	1.16	7.48	0.02	0.00
gau	4.05	0.27	0.96	0.53	0.06	11.70
hard28	–	2.40	46.98	38.98	0.36	31.50

tests, we can conclude that, among the evolutionary algorithms, HEA produces the best results. Among the meta-heuristics, CNS_BP and HEA have similar performances.

HEA is also appropriate for the solution of other types of BPP. As our next problem, we are now working on the BPP with conflict using a modified version of HEA.

Acknowledgements Open access funding provided by University of Pécs.

References

Alvim ACF, Ribeiro CC, Glover F, Aloise DJ (2004) A hybrid improvement heuristic for the one-dimensional Bin Packing Problem. J Heuristics 10(2):205–229

Borgulya I (2019) An EDA for the 2D knapsack problem with guillotine constraint. CEJOR 27(2):329–356

Bugger B, Doerner KF, Hartl RF, Reimann M: AntPacking—an ant colony optimization approach for the one-dimensional bin packing problem. In: Gottlieb J, Raidl GR (eds) EvoCOP 2004, LNCS 3004, pp 41–50 (2004)

Buljubašić M, Vasquez M (2016) Consistent neighborhood search for one-dimensional bin packing and two-dimensional vector packing. Comput Oper Res 76:12–21

Burke EK, Hyde MR, Kendall G (2006) Evolving bin packing heuristics with genetic programming. In: Parallel problem solving from nature—PPSN IX, vol 4193. Springer, Heidelberg, pp 860–869 LNCS

Cai Y, Chen H, Xu R, Shao H, Li X (2013) An estimation of distribution algorithm for the 3D Bin Packing Problem with various bin sizes. In: Yin H et al (eds) IDEAL 2013, LNCS, vol 8206, pp 401–408

Cerebio J, Irurozki E, Mendiburu A, Lozano JA (2012) A review on estimation of distribution algorithms in permutation-based combinatorial optimization problems. Prog Artif Intell 1:103–117

Coffman EG, Garey MR, Johnson DS (1996) Approximation algorithms for bin packing: a survey. In: Hochbaum D (ed) Approximation algorithms for NP-hard problems. PWS Publishing, Boston, pp 46–93

Dokeroglu T, Cosar A (2014) Optimization of one-dimensional bin packing problem with island parallel grouping genetic algorithms. Comput Ind Eng 75:176–186

Delorme M, Iori M, Martello S (2016) Bin packing and cutting stock problems: mathematical model and exact algorithms. Eur J Oper Res 255:1–20

Falkenauer E (1996) A hybrid grouping genetic algorithm for bin packing. J Heuristics 2:5–30

Fleszar K, Charalambous C (2011) Average-weight-controlled bin-oriented heuristics for the one-dimensional bin-packing problem. Eur J Oper Res 210:176–184

Fleszar K, Hindi KS (2002) New heuristics for one-dimensional bin-packing. Comput Oper Res 29:821–839

Garey MG, Johnson DS (1979) Computers and intractability. A guide to the theory of NP-completeness. Freeman, New York

Jiang H, Zhang S, Xuan J, Wu Y (2011) Frequency distribution based hyper-heuristic for the Bin-Packing Problem. In: Merz P, Hao JK (eds) EvoCOP 2011, LNCS vol, 6622. Springer, Heidelberg, pp 118–129

Kucukyilmaz T, Kiziloz HE (2018) Cooperative parallel grouping genetic algorithm for the one-dimensional bin packing problem. Comput Ind Eng 125:157–170

Loh K, Golden B, Wasil E (2008) Solving the one-dimensional bin packing problem with a weight annealing heuristic. Comput Oper Res 35:2283–2291

López-Camacho E, Terashima-Marín H, Ross P (2011) A hyper-heuristic for solving one and two-dimensional bin packing problems. In: Proceedings of the 13th annual conference companion on genetic and evolutionary computation, pp 257–258

Pelikan M, Goldberg DE, Cant´u-Paz E: BOA (1999) The Bayesian optimization algorithm. In: Proceedings of the genetic and evolutionary computation conference (GECCO-99), pp 525–532

Quiroz-Castellanos M, Cruz-Reyes L, Torres-Jimenez J, Gómez C, Héctor S, Huacuja JF, Alvim ACF (2015) A grouping genetic algorithm with controlled gene transmission for the bin packing problem. Comput Oper Res 55:52–64

Stawowy A (2008) Evolutionary based heuristic for bin packing problem. Comput Ind Eng 55:465–474

Singh A, Gupta AK (2007) Two heuristics for the one-dimensional bin-packing problem. OR Spectrum 29:765–781

Tsutsui S, Pelikan M, Goldberg DE (2003) Using edge histogram models to solve permutation problems with probabilistic model-building genetic algorithms. Technical report, IlliGAL Report No 2003022