An EDA for the 2D knapsack problem with guillotine constraint

Article *in* Central European Journal of Operations Research · June 2019

DOI: 10.1007/s10100-018-0551-x

Istvan Borgulya
University of Pecs
37 PUBLICATIONS 178 CITATIONS

An EDA for the 2D knapsack problem with guillotine constraint

István Borgulya
Faculty of Business Economics
University of Pecs

Abstract

In this paper we present an evolutionary heuristic for the 2D knapsack problem with guillotine constraint. In this problem we have a set of rectangles and there is a profit for each rectangle. The goal is to cut a subset of rectangles without overlap from a rectangular strip of width *W* and height *H*, so that the total profit of the rectangles from the subset is maximal. The sides of the rectangles are parallel to the strip sides and every cutting is restricted by orthogonal guillotine-cuts. A guillotine-cut is parallel to the horizontal or vertical side of the strip and cuts the strip into two separated rectangular strips.
Our algorithm is an estimation of distribution algorithm (EDA), where recombination and mutation evolutionary operators are replaced by probability estimation and sampling techniques. Our EDA works with two probability models. It improves the quality of the solutions with local search procedures. The algorithm was tested on well-known benchmark instances from the literature.

Keywords
Cutting-packing, knapsack, guillotine- cut, EDA.

1. Introduction

In the two-dimensional rectangular knapsack problem (2DKP) with guillotine constraint we have a set of m types of rectangles with w_i widths, h_i heights and a profit p_i where $i=1,2,...,m$. The *j*th type contains u_j rectangles. The goal is to cut a selected subset of the rectangles without overlap onto a rectangular strip of width *W* and height *H*, so that the total profit of the selected rectangles is maximal. All cuts are orthogonal guillotine-cuts and the rectangles have to be laid out orthogonally on the strip. A guillotine-cut is parallel to the horizontal or vertical side of the strip and cuts the strip into two smaller separated rectangular strips. The problem can be specified with rotation; we will regard the rectangles with fix orientation (notation: 2DKP-OG).

Based on the available number of rectangles in a type and based on the profit we can classify 2DKP. The available number of rectangles can be limited or unlimited, and the profit can be the area of the rectangle or independent from the area. In our algorithm we work with two version of 2DKP:

- The constrained unweighted version: For each rectangle i, the available number u_i is limited and the profit p_i is equal to its area. The number of types is m, $n=u_1+u_2+ \dots +u_m$.
- The constrained weighted version: For each rectangle i, the available number u_i is limited and the profit p_i is independent of its area. The number of types is m, $n=u_1+u_2+ \dots +u_m$.

The 2DKP belongs to the cutting-packing problems. It is NP-hard (Beasley 2004). Many exact, heuristic and meta-heuristic algorithms have been published to solve the 2DKP and the versions with guillotine constraint. A possible new evolutionary method for the 2DKP-OG is the estimation of distribution algorithm (EDA). 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. The new solutions replace a portion of the former population, or the population is fully replaced by the use of a probabilistic model. Therefore the EDA generates descendants without the use of recombination and mutation operations in the following two steps:

- According to selected individuals it creates, or updates a probabilistic model,
- Drawing new descendants from the distributions of the probability model (this is the sampling).

In this paper, we are interested in the EDA for 2DKP-OG. Our motivation was to build an EDA for the 2DKP-OG, which gives a better result than the earlier evolutionary techniques. For 0/1 knapsack problem there is a successful EDA (Gao et al. 2014). Our EDA for the 2DKP-OG has harder tasks: it has to select a subset of rectangles and has to organize the guillotine cuts of the selected rectangles.

From the viewpoint of our EDA the difficulties of the 2DKP-OG are the following:

- We have to select a subset of rectangles for the knapsack.
- We want to divide the strip into layers with horizontal guillotine-cuts, where the heights of the layers are not known and we do not know the best number of the layers for the optimal solution.
- We have to choose the best rectangles for every layer from the selected subset of the rectangles.
- In every layer we have to cut the rectangles with guillotine-cuts. For this we have to give the guillotine-cuttable pattern of the rectangles.

Our EDA works with two probability models. Based on the best solutions this EDA generates a probability model and based on the model it selects the subset of rectangles for cutting. Parallel this probability model the EDA generates another, second probability model and based on the second model it divides the rectangles from the subset into separated groups. Every group gets own layer that is cut by horizontal guillotine-cuts from the strip. This second probability model shows how good it is if the *i*th and the *z*th rectangles are placed into the same layer. Higher values show better pairs of rectangles in a layer. To generate a group the EDA compares pairs of rectangles based on the second model. At the end the algorithm gives the guillotine-cuttable pattern of the rectangles for the layers; this is the solution.

To improve the quality of the solutions the algorithm applies local search procedures and with given probability applies the generation of the descendent with selection and with a new mutation operator based on the second probability model too.

Our contribution, therefore, is a new hybrid EDA for the 2DKP-OG (named 2DKEDA) and its key features are the following:

- The algorithm uses two probability models to generate a descendent.
- We give a new sampling technique to select the rectangles for the layers.
- The algorithm improves the quality of the solution with the generation of a descendent with selection and a new mutation operator based on the second probability model too.

The remainder of this paper is organized as follows: Section 2 describes the related works. Section 3 defines the elements of our cutting procedure; Section 4 describes the probability model and their applications. Section 5 gives the main steps of the 2DKEDA. The computational results are reported in Section 6 and the conclusions are in Section 7.

2. Related works for 2DKP

2.1 Exact, heuristic and meta-heuristic methods

Exact methods

Some exact algorithms were published for the 2DKP and for the 2DKP-OG too. For 2DKP the exact methods are for example tree search-based algorithms (Beasley 1985b, Fekete et al. 1997, 2007), and some authors used variants of the branch and bound technique (Hadjiconstantinnou et al. 1995, Arenales et al. 1995, Caprara et al. 2004).

For 2DKP-OG we find similar methods: tree search-based algorithms (Viswanathan et al. 1993, Morabito et al. 1996, Hifi 1997), branch and bound techniques (Hadjiconstantinnou et al. 1995, Cung et al. 2000). Other methods are e.g. dynamic optimization methods (Christofides et al. 1977, Beaslay 1885a, Cintra et al. 2004) and a recursive procedure (Dolatabadi et al. 2012).

We can, however, only use these methods to solve small 2DKP cases within a reasonable period of time.

Heuristic, meta-heuristic methods

The heuristic methods are search algorithms that are able to find the global optimum only with a high degree of probability. We find construction heuristics for the 2DKP (e.g. Wu et al. 2002) and for the 2DKO-OG too (Wang 1983, Vasco 1989, Oliveira et al. 1990, Wei et al. 2015). The heuristic of Wei et al. (2015) combines the top-down and bottom-up approaches and combines rectangles into blocks. Fayard et al. (1998) published a heuristic for approximately solving the problem.

There are meta-heuristics for the problem too. There ware published for the 2DKP more genetic algorithm (GA) versions (e.g. Lai et al. 1997, Beaslay 2004, Goncalves et al. 2006), but there are simulates annealing, tabu search and GRASP algorithms too (e.g. Chen 2008, Leung et al. 2012, Alvarez-Valdes et al. 2005, Egeblad et al. 2009). For 2DKP-OG we find only a few meta-heuristics: e.g. a tabu search (Alvarez-Valdés et al. 2002) and GA versions (Parada et al. 1995, Bortfeldt et al. 2009).

2.2 Estimation of distribution algorithm

The EDAs depending on the complexity of the probability models are divided into three groups (Pelikan et al. 1999). The first group are the models without interaction in which the variables of the individual are independent from each other. Pair-wise interactions allow the second group where interactions can occur between each variable pair; and the third, the case of multivariate interactions, complicated dependencies are allowed among variables. The efficiency of the models also varies depending on the interaction too: the linear problems can solve the models of the first group, in case of the pair-wise interactions we can solve quadratic problems, while in case of the multivariate interactions we can solve complex problems.

EDAs can be categorized into three categories based on the solution representation of the problem too, i.e. discrete variables, permutation and real-valued variables. In all categories we can use different probability models according to the interaction between the variables. For example the most important models for the discrete variables are the following:

- In the models without interactions the variables of the problem are independent. Individuals may be finite bit strings, and the probability models use a probability vector. The vector gives for each bit position an estimated probability. The probability gives the estimated probability of the value 1 on the given bit position. A method in this group is the PBIL (population based incremental learning) (Baluja 1994).
- Some EDA algorithms allow pair-wise interactions among the variables. Their common feature is that the dependencies are represented with a sequence (chain) among the variables, or a tree structure represents the relations of the variables. Such variations of the methods include the MIMIC (Mutual information maximizing input clustering) (De

Bonet et al. 1996), the COMIT (Combining Optimizers with Mutual Information Trees) (Baluja et al. 1997) and BMDA (bivariate marginal distribution algorithm) (Pelikan et al. 1999).

- Models with multivariate interactions represent dependencies using either directed acyclic graphs or undirected graphs. Popular models are the Bayesian networks and the Markov networks. An example is the ECGA (extended compact genetic algorithm) (Harik et al., 1999) that groups the variables into independent clusters. Another group of methods learns Bayesian network during the evolution, which can describe even more complex dependencies among the variables. Such methods are the BOA (Bayesian optimization algorithm) (Pelikan et al. 1999), EBNA (estimation of Bayesian Networks algorithm) (Etxeberria et al. 1999) or LFDA (learning factorized distribution algorithm) (Mühlenbein et al. 1999).

There are techniques that grow the efficiency of the EDAs, too. The most important techniques are the parallelization and the hybrid EADs. In the EDAs several computational tasks can be executed in parallel: the fitness evaluation, the model building and the sampling process (see e.g. Ocenasek 2002). The hybrids EDAs usually apply local search procedures. (Detailed descriptions of the EDAs are available in Pelikan et al. 2012).

A lot of applications of the EDAs are available. Typical problems are the scalar and multi-objective optimization, timetabling, scheduling. For packing and cutting problems there are also a few applications: 3D bin packing with EDA (Cai et al. 2013), 3D strip packing with EDA (Pham 2011). For 0/1 knapsack problems there is an EDA algorithm (Gao et al. 2014). To the best of our knowledge, there is not an EDA for 2DKP with guillotine constraint.

3. Preliminaries

Our algorithm gives guillotine-cuttable pattern of rectangles and gives the cutting commands for the patterns. The important elements of our algorithm are the *rem* set, the layers, the regions with a selection procedure of the regions, the placement heuristics and a placing-cutting procedure. Let us see the details.

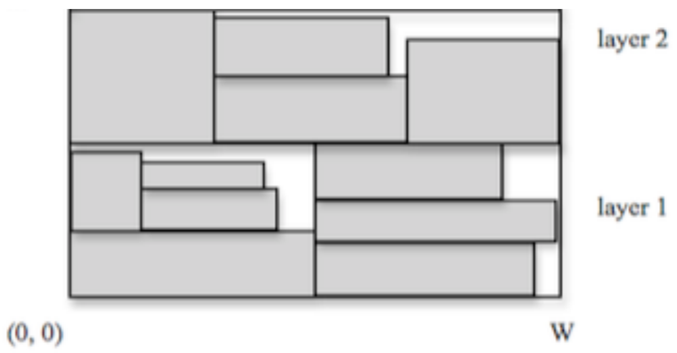


Figure 1. Layers on the strip with patterns

The rem set

Our algorithm selects rectangles for the knapsack. The unselected rectangles will store in the *rem* set.

The layers, regions and the selection of the regions

The layer is a shorter strip with the same *W* width and we can cut it with guillotine-cuts from the strip. (Figure 1 shows an example with two layers on the strip). For a layer our algorithm selects rectangles with the use of the second probability model. Let the set of the selected rectangles be Q.

The first region is the layer itself. After placing a rectangle into the bottom left-hand corner of the region there is more than one way to continue the placing process. If the width or the height of the placed rectangle is equal to the width or the height of the region, we can divide the remainder of the region horizontally or vertically and get the R1 or R2 sub-regions. Otherwise we can divide the remaining region horizontally and vertically into R1 and R2 sub-regions (see figure 2). If the width of the R2 sub-region is too small for other rectangle (see figure 2. a), we cannot use the R2 sub-region, so we only divide the region horizontally, and next we apply the recursive procedure on a remaining region. Using the regions our process *guarantees* that the rectangles will be placed without overlap and the guillotine constraint is satisfied.

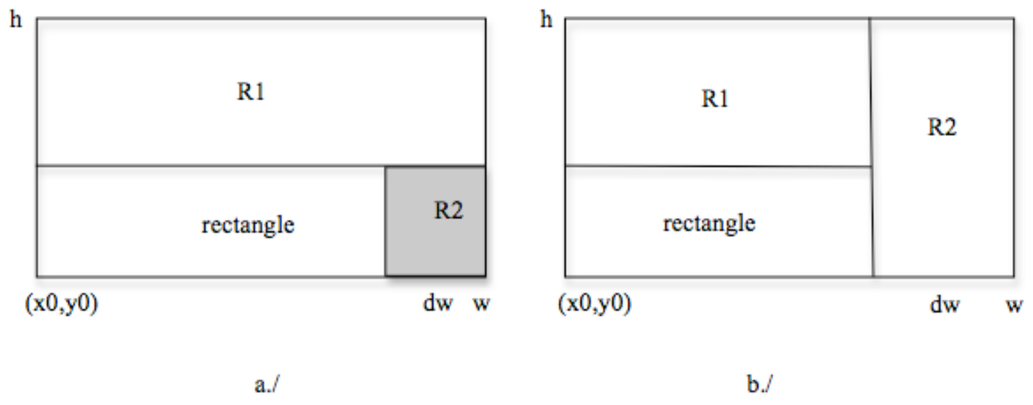


Figure 2. Two possible ways to construct the R1, R2 sub-regions. a./ dw is too small, b./ dw is good for packing.

Fit rectangle

If we can place a rectangle into the given region, it is a fit rectangle for the region. The *packlayer* procedure searches the not-yet-placed, fit rectangles for the region (from the Q set). It selects the fit rectangles and stores the first found fit rectangles into the *fit_list* vector. The length of the vector is maximum *imax (imax=min(n, 200))*. If there is no fit rectangle, the *fit_list* vector will empty and the region will empty.

A *random-fit* rectangle is a random element of the *fit_list*.

Best- fit rectangle

The *packlayer* procedure searches the *best-fit* rectangle for the region from Q too. The *best-fit* rectangle is a fit rectangle with the largest profit. For the *best-fit* rectangle it checks every rectangle from Q. If there are more *best-fit* rectangles it selects one of them with the largest width.

Fit block

If the width of the packed rectangles is smaller than the width of the region, we can apply the local search *ImpLS*. *ImpLS* works with the not-yet-packed element of *fit_list*. It builds blocks combining one, two or three rectangles one after the other from the *fit_list.* (The height of the block is the maximum height of the rectangles in the block). If *ImpLS* finds fit blocks for the empty width part of the region, it selects the fit block with the largest total profit.

Placement heuristics

We use two placement heuristics: *HP1*, *HP2*.

- *HP1*: it selects the *best-fit* rectangle for placing.
- *HP2*: it selects a *random-fit* rectangle for placing.

Placement strategy

Our placement strategy is the following: for packing it selects the *HP1* heuristic with probability p_{bf} ; otherwise it selects the *HP2* heuristic. Next it searches fit block with probability p_{imp} with the *ImpLS* local search.

The packlayer procedure

A recursive packing procedure (*packlayer*) packs rectangles on the area of the layer and gives the cutting commands for the regions and rectangles. The procedure divides the area of the layer into regions and it packs a rectangle into a region. (The packing is similar to Wei et al. 2015, but we use it with our placement strategy instead of the best fit heuristic).

The Algorithm 1 shows the main steps of *packlayer* procedure. The procedure has four parameters: *vbw* – the width of the region; *vbh* – the height of the region; *x00, y00* – the bottom left-hand corner of the region. The procedure uses the Q set of the rectangles, which have to pack, and the Q set is available for every recursive call of the procedure. The packing is happening with placement heuristics. When the procedure has completed the packing, we get the layer description. If there are remained unpacked rectangles then the layer is too small for packing all rectangles from Q.

Algorithm 1. The *packlayer* procedure.

```
packlayer(vbw,vbh,x00,y00);
        // The not-yet-packed rectangles are in Q.
        If the fit_list vector is empty then return fi
        Apply our placement_strategy.
        Let the total width and height of the packed rectangle (and fit block) be plw, plh.
        dw=vbw-plw.
        If (dw is too small width for other rectangle) then
            // placing into R1 sub-region.
                vbh=vbh-plh;y00=plh+y00;
                packlayer(vbw,vbh,x00,y00);
        else
                If (plw*(vbh-plh)<=(vbw-plw)*vbh) then
                // placing into R1 sub-region.
                        vbh=vbh-plh;y00=plh+y00;m1=vbw;vbw=plw;
                        packlayer(vbw,vbh,x00,y00);
                // placing into R2 sub-region
                        vbw=m1-plw;x00=plw+x00;vbh=plh+vbh;y00=y00-plh;
                        packlayer(vbw,vbh,x00,y00);
                else
                // placing into R2 sub-region
                        vbw=vbw-plw;x00=plw+x00;
                        packlayer(vbw,vbh,x00,y00);
                // placing into R1 sub-region.
                        vbh=vbh-plh;vbw=plw;y00=plh+y00;x00=x00-plw;
                        packlayer(vbw,vbh,x00,y00);
                fi
        fi
```

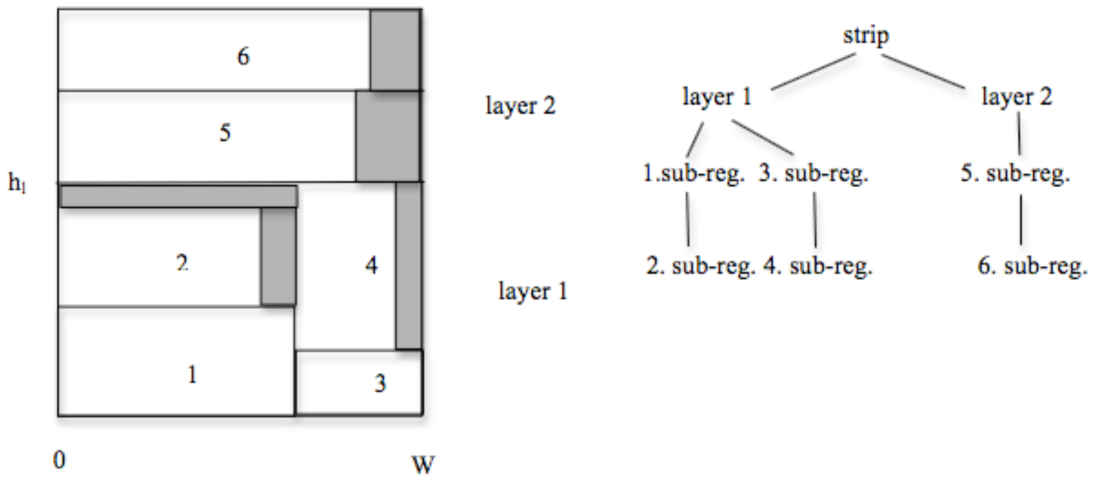
The following example demonstrates the work of the *packlayer* procedure. Recently the strip has two layers and the procedure generated 4 sub-regions in the first layer and two sub-regions in the second layer. Figure 3/a shows the strip with the two layers and the sub-regions that are identified with number. The width and height of the *i*th sub-region are (w_i, h_i) (i=1, 2, …,6). Figure 3/b shows with a tree representation the working process with the region, sub-regions.

We can generate the cutting commands based on this tree representation. Let H(x,y) and V(x,y) be the guillotine-cut commands. H(x,y) cuts the strip horizontally in the (x,y) point of the region (sub-region), V(x,y) cuts the strip vertically in the (x,y) point of the region (sub-region). In this example the cutting commands are the following:

H(0, h_l) /* cut the first layer */

$V(w_1, 0), H(0, h_1)$ /*cut the 1. sub-region */
$H(0, h_2), V(w_2, 0)$ /*cut the 2. sub-region */
$H(0, h_3)$ /*cut the 3. sub-region */
$V(w_4, 0)$ /*cut the 4. sub-region */
$H(0, h_5)$ /*cut the 5. sub-region */
$H(0, h_6)$ /*cut the 6. sub-region */

In every sub-region there are one, or more rectangles; so additional cuts would be necessary to obtain the rectangles.



a./ b./

Figure 3. Strip with sub-regions and the tree representation

Repacking local search
Usually we can repack the rectangles of the layer into a less thick layer. If we reduce the height of the layers and the *packlayer* tries to repack the rectangles, the packing can be successful (the methods in Wei et al. 2014, Cui et al. 2013, Bortfeldt et al. 2012. try to reduce the height of the layers on the basis of a similar idea). Based on this idea our local search tries to reduce the height of a layer.

Repair procedure
If the total height of the layers is too large (it is larger as H) the procedure deletes a random rectangle of a random layer and applies the *Repacking* local search. The deleted rectangles will store in the *rem* set. If the new total height is smaller as H, the improving is ready. Otherwise the procedure repeats the process; deleting another rectangle.

4. Probability models and their applications

Our EDA is different from the typical EDA methods that were showed in section 2.2. It handles the interactions among the rectangles, knapsack and layers in two levels: between the knapsack and the rectangles, and among the rectangles and the layers. So for the generation of a descendant it uses two probabilities model one after the other depending on the interactions.

4.1 Probability model for selecting rectangles for the knapsack

At every descendant the EDA selects a set of rectangles for the knapsack based on a probability model. This model is a vector named *M1* with *n* elements.

Every rectangle has a position in *M1* that stores the relative frequency of the rectangle in the knapsack. *M1* is updated periodically throughout the evolution process using some of the best performing individuals.

Let $M1_i^{gen}$ be the collected relative frequency of the *i*th rectangles until the *gen*th generation. We can update the elements of the *M1* vector

$$M1_i^{gen+1} = (1-\alpha)M1_i^{gen} + \alpha * \Delta M1_i$$

where $\Delta M1_i$ is the relative frequency of the *i*th rectangles based on the best individuals of the *gen*th generation and α denotes some relaxation factor (e.g., $\alpha=0.2$). We update *M1* periodically every *kn*th generation (e.g., $kn=10$). The computation of $\Delta M1_i$ happens as follows:

- we take the 20% best individuals from the population;
- we count in $\Delta M1_i$ how many times the *i*th rectangle is in the best individuals ($i=1,2,\ldots,n$).
- we divide the $\Delta M1_i$ vector by the number of best individuals.

Sampling M1

Sampling M1 selects a set of the rectangles for cutting from the knapsack. It selects the rectangles where

a random probability < $M1_i^{gen}$ ($i=1,2, \ldots, n$).

Let the selected rectangles be the QK set. The unselected rectangles will store in the *rem* set.

4.2 Probability model for selecting rectangles for the layers

With the first probability model we get the selected rectangles. Next we choose the best pairs of rectangles for every layer from the selected set of the rectangles. The pairs of rectangles are selected for a layer based on a second probability model. This probability model shows how good it is if the *i*th and the *z*th rectangles are cut from the same layer. Higher values show better pairs of rectangles for cutting from the same layer.

For these we use the EVL technique (see the principle of EVL in Borgulya 2006). Recently we have modified this technique the following way: we have to know the frequency of every pair of rectangles - how often they are members of the same layer in the best solutions. Let *ECM* (explicit collective memory) be an $n \times n$ matrix that stores and learns the relative frequencies of the different pairs. Every rectangle has a row and a column in the matrix. This matrix is updated throughout the evolution process using some of the best performing individuals.

Let ECM_{ij}^{gen} be the collected relative frequency of the *i*th and the *j*th rectangle (a pair) in common layers until the *gen*th generation. We can update the elements of the *ECM* matrix

$$ECM_{ij}^{gen+1} = (1-\alpha)ECM_{ij}^{gen} + \alpha\Delta ECM_{ij}$$

where ΔECM_{ij} is the relative frequency of the *i*th and the *j*th rectangles in common layers based on the best individuals of the *gen*th generation and α denotes some relaxation factor (e.g., $\alpha=0.2$). We update *ECM* periodically every *kn*th generation (e.g., $kn=10$). The computation of ΔECM_{ij} happens as follows:

- we take the 20% best individuals from the population;
- we count how many times the *i*th and *j*th rectangles are in common layers in the best individuals ($i=1,2,\ldots,n; j=1,2,\ldots,n$)
- we divide the ΔECM matrix by the number of these 'best' individuals.

We use the *ECM* matrix to estimate the probability of the good pairs of rectangles. The formula

$$pr_{ij} = \frac{ECM_{ij}^{gen}}{\sum_{t=1}^{n} ECM_{it}^{gen}}$$

gives the probability that the ith and the jth rectangles are good pair in the same layer.

Sampling ECM

Sampling ECM selects rectangles for the layers of the descendant. For every layer it selects a new Q set from the not yet selected rectangles of QK. The sampling is the following:

1. If there are unselected rectangles in QK, first it chooses an unselected rectangle randomly from QK; if there is not, go to step 3.
   This rectangle will be the first element in Q. Let this be the i*th* rectangles. For the other rectangles of Q the sampling selects the rectangles from QK, where

   a random probability $<pr_{ij}$ (the ith and jth rectangles from QK).

   If Q is ready then it calls the *packlayer* procedure, the *Repacking* procedure and after the *profitrepair* local search procedure for the layer in this order. The *profitrepair* procedure selects all the unselected rectangles of QK and all the rectangles of *rem* and checks them one after the other. The checked rectangle is swapped with one rectangle from the layer and the *Repacking* procedure is applied. The *profitrepair* accepts the move if the swap increases the total profit of the layer without increasing the height of the layer.
2. If there are unselected rectangles in QK we repeat the layer generation for a new layer (go to step 1).
3. The sampling is ready. If the total height of the layers is higher as H, it calls the *repair* procedure.

5. The 2DKEDA algorithm

5.1 The main steps of the algorithm

Our 2DKEDA generates only one descendent in every generation. First it generates the initial population. Next, it generates descendents by sampling *M1*, *ECM*. After it improves the layers of the descendent with local searches (LS).

For certain tasks, the algorithm might "get stuck" at one of the local optima. To enable escape toward 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 2 shows the main steps of 2DKEDA. The parameters of the algorithm are the following:

tmax – the maximal size of the population.
t - the first size of the population.
kn - the algorithm is controlled in every *kn*th generation.
timeend - the limit of the running time.
gp, rp - parameters of the condition of the *Restart* procedure.
LSremn- parameter of the local searches.

Algorithm 2. The main steps of 2DKEDA

```
Input: the instance, the values of the parameters.
Initial block building
Every value of the probability models M1, ECM is 0.5.
Generate the initial population.
Update the probability models M1, ECM.
```

```
    Repeat
        Do kn times
            Generate the rectangles for the knapsack by sampling M1.
            Generate the layers by sampling ECM.
            Apply local searches. Reinsertion.
        od
        If ($t<tmax$) then $t=t+1$ fi
        Apply local searches on the last descendent, reinsertion.
        Apply local searches on the best individual, reinsertion.
        Update the M1, ECM models. Restart.
    until running time>timeend
end
```

The operations and features are:

Input. The algorithm reads the instance and the values of the parameters (they are described and given in section 6.1). Every rectangle gets different identification number.

Individuals. Every individual of population P contains the description of a solution: the layers, the cutting commands and a *rem* set with *rn* elements. The *rem* set stores the rectangles that are not element of the knapsack. The individual gives all the important data of the layers (similar way as in Bortfeldt 2006) the height of the layer, the number of rectangles in the layer and the data of the rectangles: the identification number of the rectangle and the type. Next the individual stores the generated cutting commands, at end stores the element of the *rem* set (see figure 4).

Initial block building. See section 5.3.

Initial population. See section 5.3.

```
nl – numbers of layers
ht – the total height of all layers
for each layer i=1, …, nl
  $h_i$ – height of the layer
  $np_i$ – number of rectangles in the layer
  for j=1, …, $np_i$
        $t_j$ – the type of rectangle j
        $id_j$ – identification of rectangle j
     endfor
endfor
the set of the cutting commands
rn – number of the elements in rem set
the rem set
```

Figure 4. The structure of an individual

Fitness function. The fitness function of the solution is the total profit of the rectangles in the layers.

Local search. The algorithm applies three LSs (*LSrem1*, *LSrem2* and *LSrem3*). It applies the LSs one after the other (see 5.2 section).

Restart. If the fittest solution did not change in the last *gp* generations, the *Restart* procedure deletes the weakest solutions (*rp* proportion of the population).

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 or if there are fewer individuals than the size of the population (after *Restart* procedures), the new descendent is unconditionally inserted into the population until the population size is reached.

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

5.2 Local search procedures

These LSs insert rectangles from the *rem* set into the set of rectangles of the layers. There are three LSs: *LSrem1*, *LSrem2* and *LSrem3*.

LSrem1 inserts a random rectangle from *rem* into a layer. After insertion, *LSrem1* gives the possible highest height to the layer and applies the *Repacking* procedure. If the insertion improves the fitness value and the total height of the layers is not higher as *H*, it accepts the move and modifies the layer descriptions and the *rem* set. *LSrem1* makes these move on every layer.

LSrem2 and *LSrem3* work similar way as *LSrem1*, but *LSrem2* insert a random group of rectangles from *rem* into a layer, and *LSrem3* insert every rectangle from *rem* into a layer. If after the use of the *packlayer* procedure the new elements of the layer improve the fitness value and the total height of the layers is not higher as *H*, it accepts the moves and modifies the layer descriptions and the *rem* set.

We apply the LSs one after the other: *LSrem1*+ *LSrem2* + *LSrem3* and repeat the group *LSremn* times.

5.3 Details of the implementation

When describing the algorithm, some heuristic solutions were not described. Let us see them now one-by-one.

Initial block building

The algorithm can build blocks if there is a type with more rectangles. Using the initial blocks the algorithm can run faster.

It works with the blocks as new rectangles and defines new rectangle-types for the blocks (width and height). At every type with more elements the algorithm builds blocks with p_{block} probability. The blocks can build from two or four rectangles of the same type. After the initial block building an additional sub-type shows the block structure (notation: btype=1, or 2, or 3, or 4) at every type. The values of the *i*th sub-type are the following:

1. There is not block, the type reminds the initial input type. *btype=1*
2. If the number of rectangles of the *i*th type is more than three, it builds larger rectangles with $2*w_i$ width and $2*h_i$ height if we can place the blocks onto the strip. The algorithm builds all possible blocks and the blocks get a new number of type. *btype=4.*
3. If there are remained elements after the second step and the number of remained rectangles of the *i*th type is more than two, with 0.5 probability it builds larger rectangles with $2*w_i$ width and h_i height (otherwise with w_i width and $2*h_i$ height) if we can place the blocks onto the strip. The algorithm builds all possible blocks and the blocks get a new number of type. *btype=2* (or *btype=3).*

At the end the algorithm updates the number of the types and all rectangles get identification numbers.

Initial population.

The initial population we can generate by sampling *M1* and *ECM*. But based on our test results, we can improve the quality of the initial population the following random generation:

For every individual we first give the set of all the rectangles and call the *packlayer* procedure to pack the first layer, where the height of the layer is a random height from the interval [*H*0.3, H-1*]. If there are remained unpacked rectangles, we generate the next layer: we give the set of the remained rectangles and call the *packlayer* procedure to pack the next layer with the same height. We repeat the process if the total height of the layers is not higher as *H*. If it is necessary at end it applies the *repair* procedure. The unselected rectangles will store in the *rem* set.

Mutation based on the ECM model

With the use of the two probabilities models we can generate descendent in our EDA. Recently we get the descendent in two steps: sampling *M1* and sampling *ECM*. In an evolutionary algorithm usually we generate the descendent in the following steps: selection of the parents, with recombination of the parents we get the descendent and after with mutation we modify the new descendent. Recently we propose that instead of recombination and mutation we can use only mutation based on the *ECM* probability model.

Let the new mutation be a swap of rectangles between two layers in the descendent. The mutation is based on the *ECM*, so it selects the *i*th and the *j*th rectangles from the given layer with the largest pr_{ij} and selects randomly another *k*th rectangles from the layer (the number of the rectangles in the layer ≥3). If the *i*th and *k*th rectangles are in the same layer of the best individual it do not make swap. Otherwise it chooses the *z*th rectangle from other layer with the largest pr_{iz} probability and swaps the *k*th, *z*th rectangles between the layers.

We can use this new mutation operator: we can generate the descendent with truncation selection and mutation based on the *ECM*. The quality of the descendent is good (see the test results in section 6.1), so we generate the descendent two different ways in our algorithm. With p_{samp} probability the algorithm applies the sampling *M1* and *ECM*, otherwise it applies the selection and mutation:

- *Selection operator*. The algorithm selects an individual based on truncation selection. In this selection, only the best *tp* percentage of the population is considered a potential parent.
- *Mutation operators*. The mutation is swap of rectangles between two layers based on the *ECM* model. It repeats the swaps three times and after applies *packlayer*, *repacking* for the layers. If the total height of the layers is too large, it applies the *repair* procedure.

6. Experimental results

The 2DKEDA algorithm was implemented in C++. It was executed on an 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 with benchmark instances that are used generally in publications. The instance sets available e.g.: http://www.computational-logistics.org/orlib/topic/2dkpp-gcut/index.html.

The constrained unweighted instances are the following:

- set1 consists 46 instance. In these instances *m* ranges from 10 to 56 and *n* ranges from 18 to 258. The names of the instances are: OF1, OF2, W, CU1-CU11, 2s, 3s, A1s, A2s, CHL1s-CHL4s, CHL5-CHL7, Hchl3s-Hchl8s, A3-A5, APT30-APT39.
- set2 consists 13 instances. In these instances *m* ranges from 10 to 50 and *n* ranges from 10 to 50. The names of the instances are: gcut1-gcut13.
- set3 with 21 instances. In these instances *m* and *n* range from 16 to 197. The names of the instances are: C11-C73.

The constrained weighted instances are:

- set5 consists 36 instances. In these instances *m* ranges from 10 to 60 and *n* ranges from 19 to 325. The names of the instances are: CHW1, CHW2, CW1-CW11, 2, 3, A1, A2, STS2, STS4, Hchl1, Hchl2, Hchl9, APT40-APT49.
- set6 consists 21 instances. In these instances *m* ranges from 5 to 33 and *n* ranges from 7 to 97. The names of the instances are: ngcut-ngcut12, hccut03, hccut08, wang20, cgcut03, okp1-okp5.
- set7 consists 630 instances. The name of the instance set is ngcutfs. There are tree problem types (subsets in set7: ngcutfs1, ngcutfs2 and ngcutfs3). In these instances *m* is varied *m*= 40, 50, 100, 150, 250, 500, 1000 and *n* ranges from 40 to 4000. In every problem type for every *m* the number of rectangles per instance is $n=m*Q$, where Q={*1, 3,4*}.

6.1 Parameter selection

We analysed the process of 2DKEDA to determine how the parameter values affect the convergence. From the 767 test instances we chose 30 instances for the parameter selection. They are the first 5 instance groups of the set3, the first 10 instances of the ngcutfs2 instance group and the APT40-APT44 instances of the set5 data set.

Because our algorithm has similar structure and parameters as our earlier algorithm had in (Borgulya 2014), we could accept the earlier parameter values. These parameters are the population size (*t* and *tmax* parameter), the frequency of checks (*kn* parameter), the generation in the first stage (*itt* parameter), the parameters of the *Restart* procedures (*gp* and *rp*) and of the truncation selection (*tp*). The accepted parameter values are the following: *t=5*, *tmax=30*, *itt=5*, *kn=5*, *gp=300*, *rp=0.7* and *tp=0.1*.

The parameter values of the initial block building, the *fit block* search, the LSs and of the sampling are new parameters in 2DKEDA. These parameters are the p_{block}, p_{imp}, *LSremn* and p_{samp}.

- For the value of the p_{block} we analyzed different values: 0, 0.1, 0.2,…, 0.9, 1. We got the average best result at 0.5 probability.
- For the probability of the *fit block* search we analysed the 0, 0.25 and 0.5 values. We got the average best result at p_{imp} =0 or at p_{imp} =0.25 probabilities depending on the instances.
- *LSremn* is an important parameter: with the use of the LSs we can improve significantly the quality of the result. The algorithm gives with more than 15-20% better results if we use the LSs.
  The LS parameter depends on the instances too. We analysed different values for *LSremn,* and found more appropriate values based on the average best results. At the end we choose the following values: if *n* is fewer than 100, *LSremn* =1000 or 2000; if it is bigger than 100, *LSremn* =20 or 200.
- 2DKEDA can generate the descendant with sampling or with selection and mutation. For the value of the p_{samp} we analyzed different values: 1, 0.75, 0.5, 0.25 and 0. We can see the results in Table 1. The table gives the average best results and the average results on the selected instances for the parameter selection. We got the best results of set5 at p_{samp} =0 probability, and the best results of set3 at p_{samp} =1 probability. So the application of the selection and mutation improved the best result of set5 with 0.6-2.5 %. At set3 we got the second best result at p_{samp} =0 probability, too. The best results of set7 are also at p_{samp} =0 probability, but the differences in the results are not large; we can use other p_{samp} probabilities as well.
  Because the best p_{samp} probability depends on the instances we decided to use both p_{samp} =1 and p_{samp} =0. During the test the algorithm ran 10 times on each test instance. We modified to use of the p_{samp} parameter during the 10 runs the following way: p_{samp} =1 in 5 runs and p_{samp} =0 in 5 runs.

Table 1. Results (gap %) on the selected instances

p_{samp}	set5		set3		set7	
	av. best	average	av. best	average	av. best	average
1	5.63	6.77	**1.16**	**2.86**	0.91	1.07
0.75	5.46	6.62	2.09	3.33	0.89	1.25
0.50	5.31	6.09	2.30	3.09	0.89	0.95
0.25	3.78	5.88	2.81	3.22	0.89	1.02
0	**3.10**	**4.97**	2.02	3.11	**0.86**	**0.90**

For the time limit we found different values in the papers: the method of Wei et al. (2015) allowed duration of 120 CPU seconds for each test problem (except the gcut13 from set2, where the time limit was 365 second), but the other methods (e.g. Bortfeld et al. 2009) did not give a time limit. We allowed duration of 300 CPU seconds for each test problem.

6.2 Computation experience

2DKEDA was run 10 times on each test instance of the test sets, and we provide the best results for every instance or as gap%, which is the percentage gap to the profit upper bound (or optimum) (notation UB), namely, gap% = 100 *(UB - obtained solution)/UB. The results on the test sets available in the appendix.

The test results show that the average best results in gap% are fewer as 1% on set1, set2, set5 and set6; in the case of set3 and set7 the gap% is 1.37 and 1.11 respectively. The algorithm provided the best results on set2 and set6 and on the large instances of set7. On set2 and set6 the gap% is 0.13 and 0.15 respectively and in both cases there were only two instances where the algorithm did not find the optimal solutions. On the set7 if $m \geq 500$ or $n \geq 1000$ the gap% is between 0 and 0.08 and in 75% of these instances it managed to find the optimal solutions. The best results show that the success of 2DKEDA is not instance-size dependent.

Figure 5 and figure 6 show the convergence behaviour of the algorithm. Figure 5 shows the results of the C5, C6 and C7 instance groups from set3. They are medium and large instances: $72 < n < 198$. 2DKEDA ran 500 CPU seconds on each instance and the figure shows the average best results (in gap%) at the end of 5, 10, 30, 60, 120, 240, 360 and 500 CPU seconds. The curves show that 2DKEDA improves the results continuously; so we can increase the likelihood of finding better solutions by choosing longer running times.

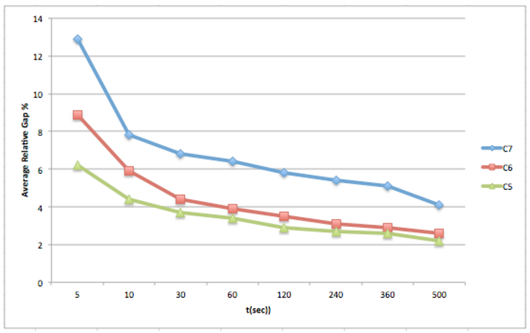


Figure 5. Convergence behaviour of 2DKEDA on set3

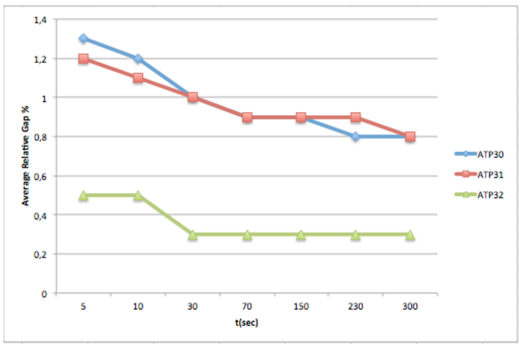


Figure 6. Convergence behaviour of 2DKEDA on set1

Figure 6 shows the results of the ATP30, ATP31 and ATP32 instances from set1. They are large instances: $191 < n < 259$. The running time was 300 CPU seconds on each instance and the figure shows the average best results (in gap%) at the end of 5, 10, 30, 70, 150, 230 and 300 CPU seconds. The algorithm is more effective with these instances than the C5, C6 and

C7 from set3. In the first 5 seconds the gap% are smaller than 1.4% and slowly descends to 0.3 -0.8%. Because the improvement of the results is very slow, a longer running time cannot help finding better solutions.

Our goal was to build an estimation of distribution algorithm for the problem, which gives better result than the earlier evolutionary techniques. For 2DKP-OG we find only a few meta-heuristics, evolutionary methods. So for comparison we can chose only the CLGAL from Bortfeldt et al. (2009) that is a GA and one of the best heuristics for the problem. The CLGAL was executed on Intel PC Core2 at 3 GHz processor with 2 GB RAM. It was implemented in C, and the authors did not give a time limit. In our comparison we give the results of IBBA from Wei et al. (2015) too, that is recently the best heuristic for the problem. The IBBA was executed on an Intel Xeon E5430 clocked at 2.66 GHz (Quad Core) with 8 GB RAM running the CentOS 5 linux operating system. It was implemented in C++ and the time limit is set to 120 seconds for each instance.

For comparison we show the results based on Wei et al. (2015). The comparison between CLGAL, IBBA and 2DKEDA is summarized in Table 2. This table gives the average best result in gap%, the names of the sets, the methods, the number of instances in the sets (#inst) and the number of instances where optimal solutions were found (#opt). On the set1, set2 and set5 the results of CLGAL were not published. In the comparison of 2DKEDA and CLGAL, we see that CLGAL has only on the set3 better result with 0.16 gap%. On the set6 2DKEDA is better with 1.1 gap% and on the subsets of set7 2DKEDA has better results with 0.11 – 0.22 gap%. Comparing the number of the optimal solutions found shows that 2DKEDA has better results on set3, set6 and set7 as well.

In the comparison of 2DKEDA and IBBA we see that every result of IBBA is better. IBBA has better results with 0.1 – 0.8 gap% and found more optimal solutions, too. The results of the algorithms are similar only at set2. We can compare the running times of IBBA and 2DKEDA, too. 2DKEDA use about two times more running times than IBBA.

Table 2. Comparison of the average best results (gap%) of the methods

sets	**#inst**	**CLGAL**		**2DKEDA**		**IBBA**	
		#opt	**gap%**	**#opt**	**gap%**	**#opt**	**gap%**
set1	46		-	15	0.43	**46**	**0**
set2	13		-	11	**0.13**	**12**	**0.13**
set3	21	5	1.21	6	1.37	**7**	**0.54**
set5	36		-	18	0.95	**34**	**0.15**
set6	21	14	1.25	19	0.15	**21**	**0**
set7	630	194	1.26	196	1.11	**315**	**0.92**
-ngcutfs1	210	63	1.23	71	1.01	**-**	**0.92**
-ngcutfs2	210	64	1.34	56	1.24	**-**	**1.00**
-ngcutfs3	210	67	1.21	69	1.09	**-**	**0.83**

We can conclude that 2DKEDA is better evolutionary method for the problem as CLGAL based on the comparison.

7. Conclusion

In this paper we have presented an estimation of distribution algorithm for the 2D knapsack problem with guillotine constraint. Our algorithm uses two probability models to generate a descendent. Based on the first model it selects the subset of rectangles for cutting and based on the second model it divides the rectangles from the subset into separated layers. We give a new sampling technique to select the rectangles for the layers. The algorithm improves the quality of the solution with the generation of a descendent with selection and a new mutation operator based on the second probability model too.

Our goal was to build an estimation of distribution algorithm for the problem, which gives better result than the earlier evolutionary technique. For comparison we can choose only the CLGAL from Bortfeldt et al. (2009), which is one of the best heuristics for the problem. The comparison between CLGAL and our algorithm show that our evolutionary algorithm has better results generally.

8. References

Alvarez-Valdes R, Parajón A, Tamarit JM (2002) A tabu search algorithm for large-scale guillotine (un)constrained two-dimensional cutting problems. Computers and Operations Research 29, 925–947.

Alvarez-Valdes R, Parreño F, Tamarit JM (2005) A GRASP algorithm for constrained two-dimensional non-guillotine cutting problems. Journal of the Operational Research Society 56, 414–425.

Arenales M, Morabito R (1995) An AND/OR-graph approach to the solution of two dimensional non-guillotine cutting problems. European Journal of Operational Research 84, 599–617.

Baluja S (1994) Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning. Tech. Rep. No. CMU-CS-94-163, Carnegie Mellon University, Pittsburgh, PA.

Baluja S, Davies S (1997) Using optimal dependency-trees for combinatorial optimization: Learning the structure of the search space. Proceedings of the International Conference on Machine Learning, pp. 30–38.

Beasley JE (1985a) Algorithms for unconstrained two-dimensional guillotine cutting. Journal of the Operational Research Society, 36, 297–306.

Beasley JE (1985b) An exact two-dimensional non-guillotine cutting tree search procedure. Operations Research, 33, 49–64.

Beasley JE (2004) A population heuristic for constrained two-dimensional non-guillotine cutting. European Journal of Operational Research, 156, 601–627.

Borgulya I (2006) An Evolutionary Algorithm for the biobjective QAP. In: Reusch B (ed) Computational Intelligence, Theory and Applications „Advances in Soft Computing", Springer series, 577-586.

Borgulya I (2014) A Parallel Hyper-Heuristic Approach for the Two-Dimensional Rectangular Strip-Packing Problem. Journal of Computing and Information Technology 22.4: 251-266.

Bortfeldt A (2006) A genetic algorithm for the two-dimensional strip packing problem with rectangular pieces. European Journal of Operational Research, 172, 814–837.

Bortfeldt A, Jungmann S (2012) A tree search algorithm for solving the multi- dimensional strip packing problem with guillotine cutting constraint. Annals of Operations Research, 196, 53–71.

Bortfeldt A, Winter T (2009) A genetic algorithm for the two-dimensional knapsack problem with rectangular pieces. International Transactions in Operational Research, 16, 685–713.

Burke EK, Hyde MR, Kendall G, Woodward J (2012) Automating the packing heuristic design process with genetic programming - Evolutionary computation, MIT Press

Cai Y, Chen H, Yu R, Shao H, Li Y (2013) An estimation of Distribution Algorithm for the 3D Bin Packing Problem with Various Bin Sizes

Caprara A, Monaci M (2004) On the 2-dimensional knapsack problem. Operations Research Letters 32, 5–14.

Chen Y (2008) A recursive algorithm for constrained two-dimensional cutting problems. Computational Optimization and Applications, 41, 337–348.

Christofides N, Whitlock C (1977) An algorithm for two-dimensional cutting problems. Oper. Res. 25(1), 30–44.

Cintra G, Wakabayashi Y (2004) Dynamic Programming and Column Generation Based Approaches for Two- Dimensional Guillotine Cutting Problems. In: Ribeiro CC, Martins SL (Eds.) WEA 2004, LNCS 3059, 175– 190, 2004.

Cui Y, Yang L, Chen Q (2013) Heuristic for the rectangular strip packing problem with rotation of items. Computers & Operations Research 40. 1094–1099.

Cung V, Hifi M, Le Cun B (2000) Constrained two-dimensional guillotine cutting stock problems: A best-first branch-and-bound algorithm. International Transactions in Operational Research, 7, 185–201.

De Bonet JS, Isbell CL, Viola JrP (1996) MIMIC: Finding Optima by Estimating Probability Densities Advances in Neural Information Processing Systems 9, NIPS, Denver, CO, USA, December 2-5, 1996

Dolatabadi M, Lodi A, Monaci M (2012) Exact algorithms for the two-dimensional guillotine knapsack. Computers & Operations Research, 39, 48–53.

Egeblad J, Pisinger D (2009) Heuristic approaches for the two- and three-dimensional knapsack packing problem. Computers and Operations Research 36(4): 1026–49.

Etxeberria R, Larranaga P (1999) Global optimization using Bayesian networks. In: Rodriguez MR, Ortiz S, Hermida RS (Eds) Second Symposium on Artificial Intelligence (CIMAF-99), Habana, Cuba, 1999. pp 332–339.

Fayard D, Hifi M, Zissimopoulos V (1998) An efficient approach for large-scale two-dimensional guillotine cutting stock problems. Journal of the Operational Research Society, 49, 1270–1277.

Fekete SP, Schepers J (1997) On more-dimensional packing III: Exact algorithms. Technical Report ZPR97-290, Mathematisches Institut, Universität zu Köln.

Fekete SP, Schepers J, van der Veen JC (2007) An Exact Algorithm for Higher-Dimensional Orthogonal Packing. Operations Research 55, 569–587.

Gao S, Qiu L, Cungen C (2014) Estimation of Distribution Algorithms for Knapsack Problem. Journal of Software, Vol. 9, no. 1 pp. 104-110.

Gonçalves JF, Resende MGC (2006) A hybrid heuristic for the constrained two-dimensional non-guillotine orthogonal cutting problem. AT&T Labs Research Technical report TD-&UNQN6.

Hadjiconstantinou E, Christofides N (1995) An exact algorithm for general, orthogonal, two-dimensional knapsack problems. European Journal for Operational Research 83, 39–56.

Harik G (1999) Linkage learning via Probabilistic Modeling in the ECGA. IlliGAL technical report Illinois Genetic Algorithms Laboratory Urbana

Hifi M (1997) An improvement of Viswanathan and Bagchi's exact algorithm for constrained two-dimensional cutting stock. Computers and Operations Research 24 (8), 727–736.

Lai KK, Chan JWM (1997) An evolutionary algorithm for the rectangular cutting stock problem. International Journal of Industrial Engineering 4, 130–139.

Leung SCH, Zhang D, Zhou C, Wu T (2012) A hybrid simulated annealing metaheuristic algorithm for the two-dimensional knapsack packing problem Computers & Operations Research 39 (2012) 64–73

Morabito R, Arenales M (1996) Staged and constrained two-dimensional guillotine cutting problems: an AND/OR- graph approach. European Journal of Operational Research 94, 548–560.

Mühlenbein H, Mahnig T (1999) FDA – A scalable evolutionary algorithm for the optimization of additively decomposed functions. Evolutionary Computation, 7(4):353–376.

Ocenasek J (2002) Parallel Estimation of Distribution Algorithms. PhD Thesis Brno University of Technology.

Oliveira JF, Ferreira JS (1990) An improved version of Wang's algorithm for two-dimensional cutting problems. European Journal of Operational Research 44, 256–266.

Parada V, Munoz R, Gomes A (1995) A Hybrid Genetic Algorithm for the Two-Dimensional Cutting Problem. In: Biethahn J, Nissen V (Eds.) Evolutionary Algorithms in Management Applications. Springer, Berlin 1995.

Pelikan M, Goldberg DE, Cant´u-Paz E (1999) BOA: The Bayesian optimization algorithm. Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-99), 525–532.

Pelikan M, Hauschild MW, Lobo FG (2012) Introduction to Estimation of Distribution Algorithms Medal report No. 2012003 University of Missouri-St. Louis.
Pham N (2011) Investigations of constructive approaches for examination timetabling and 3D-strip packing. PhD thesis, University of Nottingham.
Vasco FJ (1989) A computational improvement to Wangs's two-dimensional cutting stock algorithm. Computers and Industrial Engineering 16(1), 109–115.
Viswanathan KV, Bagchi A (1993) Best-first search methods for constrained two-dimensional cutting stock problems. Operation Research 41(4), 768–776.
Wang PY (1983) Two Algorithms for Constrained Two-Dimensional Cutting Stock Problems. Operation Research 31, 573–586.
Wei L, Tian T, Zhu W, Lim A (2014) A block-based layer building approach for the 2D guillotine strip packing problem. European Journal of Operational Research. 239, 58-69.
Wei L, Lim A (2015) A bidirectional building approach for the 2D constrained guillotine knapsack packing problem. European Journal of Operational Research. 242, 63-71.
Wu YL, Huang W, Lau SC, Wong CK, Young GH (2002) An effective quasi-human based heuristic for solving the rectangle packing problem. European Journal of Operational Research 41, 341–358.

Appendix.

In the appendix the tables give the results of the test sets. In the tables we see the names of the instances or instance groups, the number of types of rectangles (*m*) the number of rectangles (*n*), the optimum or upper bound (*opt/upper*), the number of instances where the optimal solutions were found in an instance group *(#opt*) or the number of optimal solutions found at an instance in 10 run (*Hits*), the average best result in gap% of an instance group or the best profit found at an instance.

Table 3. The results of 2DKEDA on set1.

Inst.	m	n	Opt/upper	Hits	best profit	Inst.	m	n	Opt/upper	Hits	best profit
ATP30	38	192	140904	0	140672	CU11	50	134	924696	0	921089
ATP31	51	258	823976	0	819253	2s	10	23	2778	0	2740
ATP32	56	249	38068	0	37990	3s	20	62	2721	0	2694
ATP33	44	224	236611	0	236549	A1s	20	62	2950	3	2950
ATP34	27	130	361167	0	358840	A2s	20	53	3535	0	3466
ATP35	29	153	621021	0	617400	STS2s	30	78	4653	5	4653
ATP36	28	153	130744	0	130092	STS4s	20	50	9770	0	9700
ATP37	43	222	387118	0	385811	CHL1s	30	63	13099	0	13036
ATP38	40	202	261395	0	260536	CHL2s	10	19	3279	2	3279
ATP39	33	163	268750	0	266980	CHL3s	15	35	7402	10	7402
OF1	10	23	2737	0	2713	CHL4s	15	27	13932	6	13932
OF2	10	24	2690	10	2690	CHL5	10	18	390	2	390
W	20	62	2721	4	2721	CHL6	30	65	16869	0	16683
CU1	25	82	12330	10	12330	CHL7	35	75	16881	0	16751
CU2	35	90	26100	0	26100	Hchl3s	10	51	12215	0	12209
CU3	45	158	16723	0	16679	Hchl4s'	10	32	11994	0	11960
CU4	45	113	99495	2	99495	Hchl5s'	25	60	45361	0	44893
CU5	50	120	173364	0	172942	Hchl6s	22	60	61040	0	60828
CU6	45	124	158572	6	158572	Hchl7s	40	90	63112	0	62797
CU7	25	56	247150	2	247150	Hchl8s	10	18	911	0	894
CU8	35	78	433331	0	433310	A3	20	46	5451	2	5451
CU9	25	76	657055	10	657055	A4	20	35	6179	0	6083
CU10	40	129	773772	0	772892	A5	20	45	12985	0	12961

Table 4-5. The results of 2DKEDA on set2 and set3.

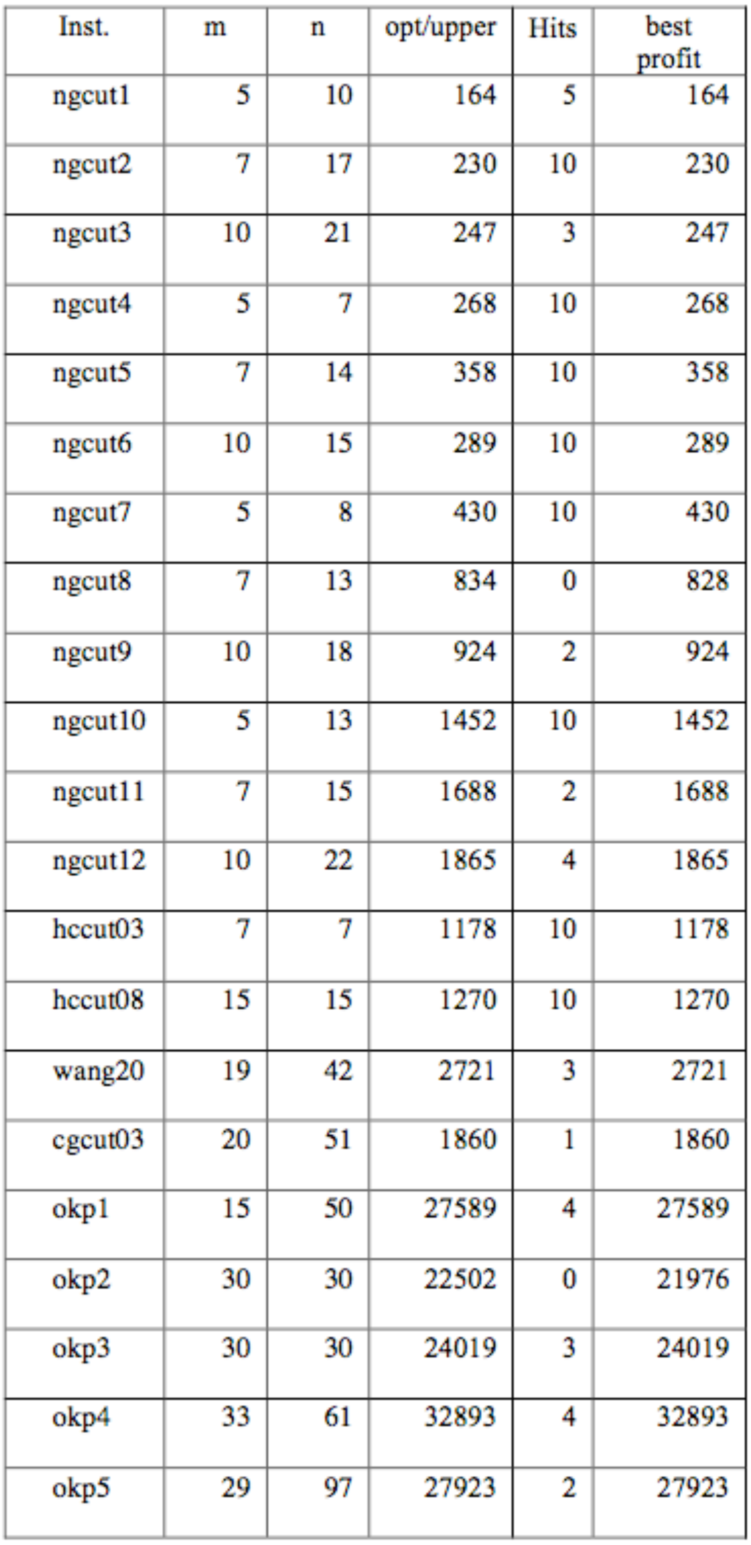
Inst.	m	n	opt/upper	Hits	best profit
gcut01	10	10	48368	8	48368
gcut02	20	20	59307	6	59307
gcut03	30	30	60241	1	60241
gcut04	50	50	60942	0	60925
gcut05	10	10	195582	10	195582
gcut06	20	20	236305	3	236305
gcut07	30	30	238974	5	238974
gcut08	50	50	245758	6	245758
gcut09	10	10	919476	6	919476
gcut10	20	20	903435	5	903435
gcut11	30	30	955389	7	955389
gcut12	50	50	970744	6	970744
gcut13	32	32	8736757	0	8591332

Inst.	m	n	opt/upper	Hits	best profit
C11	16	16	400	10	400
C12	16	16	400	2	400
C13	17	17	400	0	385
C21	25	25	600	2	600
C22	25	25	600	0	596
C23	25	25	600	5	600
C31	28	28	1800	2	1800
C32	28	28	1800	5	1800
C33	29	29	1800	0	1760
C41	49	49	3600	0	3541
C42	49	49	3600	0	3521
C43	49	49	3600	0	3563
C51	73	73	5400	0	5342
C52	73	73	5400	0	5301
C53	73	73	5400	0	5322
C61	97	97	9600	0	9395
C62	97	97	9600	0	9463
C63	97	97	9600	0	9451
C71	196	196	38400	0	37470
C72	196	196	38400	0	37442
C73	197	197	38400	0	37277

Table 6. The results of 2DKEDA on set5.

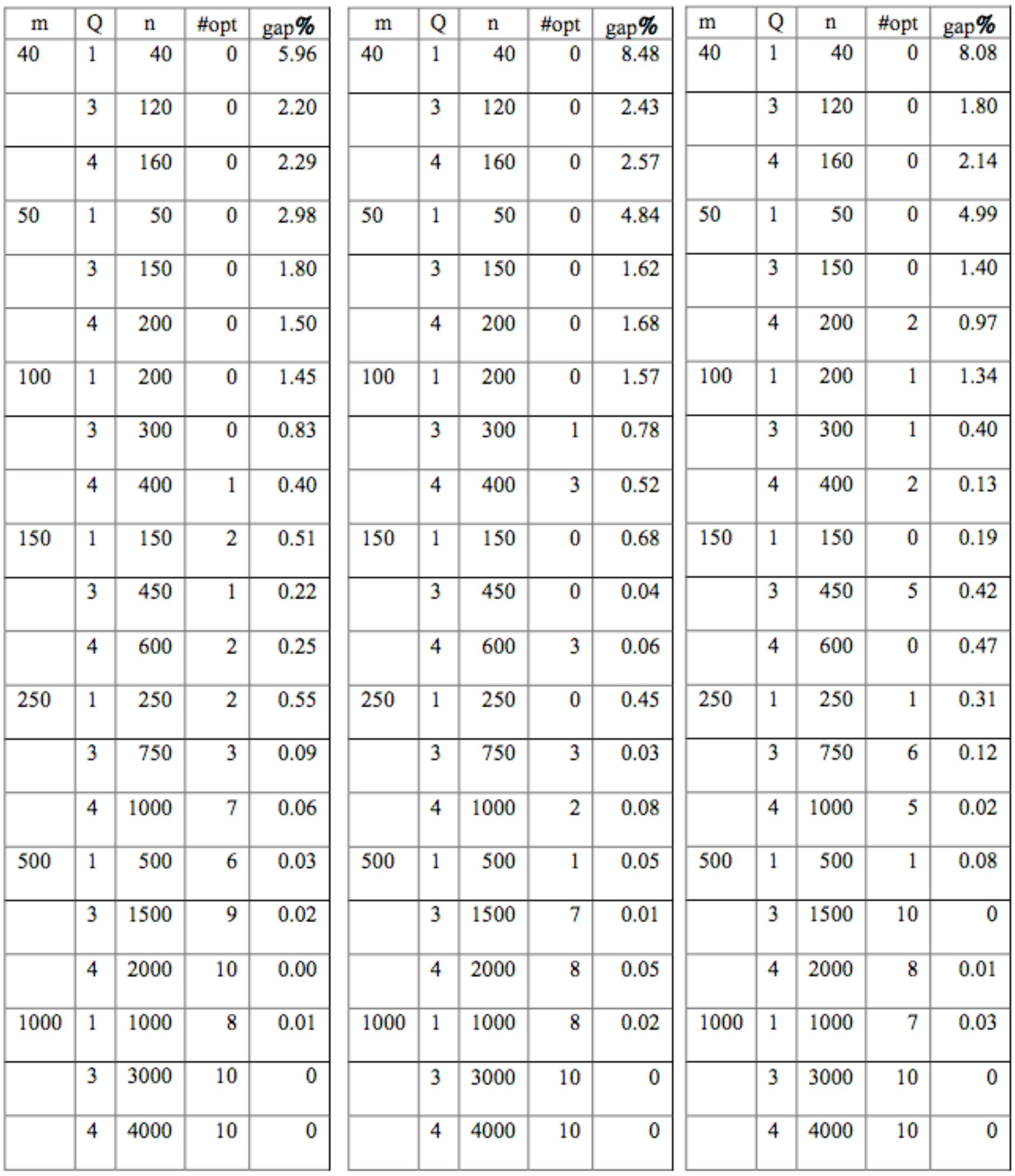
Inst.	m	n	opt/upper	Hits	best profit	Inst.	m	n	opt/upper	Hits	best profit
ATP40	56	290	67154	0	66245	2	10	23	2892	0	2856
ATP41	36	177	206542	0	204123	3	20	62	1860	3	1860
ATP42	59	325	34098	0	32925	A1	20	62	2020	0	1940
ATP43	49	259	222570	0	212062	A2	20	53	2505	0	2455
ATP44	39	196	73868	0	72348	STS2	30	78	4620	10	4620
ATP45	33	156	74691	2	74691	STS4	20	50	9700	2	9700
ATP46	42	197	149911	0	148026	CHL1’	30	63	8671	0	8551
ATP47	43	204	150234	0	147290	CHL2	10	19	2326	10	2326
ATP48	34	167	167660	0	165419	CHL3	15	35	5283	10	5283
ATP49	25	119	219354	0	211796	CHL4	15	27	8998	10	8998
CHW1	10	23	2892	0	2856	Hchl1	30	65	11303	0	11142
CHW2	20	62	1860	3	1860	Hchl2	35	75	9954	0	9797
CW1	25	67	6402	10	6402	Hchl9	35	76	5240	2	5240
CW2	35	63	5354	2	5354						
CW3	40	96	5689	5	5689						
CW4	39	86	6175	2	6175						
CW5	35	91	11659	0	11580						
CW6	55	149	12923	10	12923						
CW7	45	123	9898	10	9898						
CW8	60	168	4605	0	4504						
CW9	50	131	10748	10	10748						
CW10	60	130	6515	10	6515						
CW11	60	114	6321	5	6321						

Table 7. The results of 2DKEDA on set6.



Inst.	m	n	opt/upper	Hits	best profit
ngcut1	5	10	164	5	164
ngcut2	7	17	230	10	230
ngcut3	10	21	247	3	247
ngcut4	5	7	268	10	268
ngcut5	7	14	358	10	358
ngcut6	10	15	289	10	289
ngcut7	5	8	430	10	430
ngcut8	7	13	834	0	828
ngcut9	10	18	924	2	924
ngcut10	5	13	1452	10	1452
ngcut11	7	15	1688	2	1688
ngcut12	10	22	1865	4	1865
hccut03	7	7	1178	10	1178
hccut08	15	15	1270	10	1270
wang20	19	42	2721	3	2721
cgcut03	20	51	1860	1	1860
okp1	15	50	27589	4	27589
okp2	30	30	22502	0	21976
okp3	30	30	24019	3	24019
okp4	33	61	32893	4	32893
okp5	29	97	27923	2	27923

Table 8-10. The average best results (gap %) of 2DKEDA on set7 (ngcutfs1, ngcutfs2 and ngcutfs3).



m	Q	n	#opt	gap%
40	1	40	0	5.96
	3	120	0	2.20
	4	160	0	2.29
50	1	50	0	2.98
	3	150	0	1.80
	4	200	0	1.50
100	1	200	0	1.45
	3	300	0	0.83
	4	400	1	0.40
150	1	150	2	0.51
	3	450	1	0.22
	4	600	2	0.25
250	1	250	2	0.55
	3	750	3	0.09
	4	1000	7	0.06
500	1	500	6	0.03
	3	1500	9	0.02
	4	2000	10	0.00
1000	1	1000	8	0.01
	3	3000	10	0
	4	4000	10	0

m	Q	n	#opt	gap%
40	1	40	0	8.48
	3	120	0	2.43
	4	160	0	2.57
50	1	50	0	4.84
	3	150	0	1.62
	4	200	0	1.68
100	1	200	0	1.57
	3	300	1	0.78
	4	400	3	0.52
150	1	150	0	0.68
	3	450	0	0.04
	4	600	3	0.06
250	1	250	0	0.45
	3	750	3	0.03
	4	1000	2	0.08
500	1	500	1	0.05
	3	1500	7	0.01
	4	2000	8	0.05
1000	1	1000	8	0.02
	3	3000	10	0
	4	4000	10	0

m	Q	n	#opt	gap%
40	1	40	0	8.08
	3	120	0	1.80
	4	160	0	2.14
50	1	50	0	4.99
	3	150	0	1.40
	4	200	2	0.97
100	1	200	1	1.34
	3	300	1	0.40
	4	400	2	0.13
150	1	150	0	0.19
	3	450	5	0.42
	4	600	0	0.47
250	1	250	1	0.31
	3	750	6	0.12
	4	1000	5	0.02
500	1	500	1	0.08
	3	1500	10	0
	4	2000	8	0.01
1000	1	1000	7	0.03
	3	3000	10	0
	4	4000	10	0