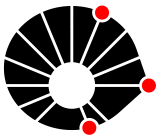
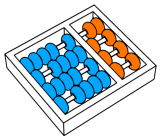




Heuristic and Exact Algorithms for the 2D Knapsack Problem with Relations Between Items

K. Rollmann *W. Cardoso* *V. L. Lima* *F. K. Miyazawa*

Relatório Técnico - IC-PFG-17-13
Projeto Final de Graduação
2017 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Contents

Heuristic and Exact Algorithms for the 2D Knapsack Problem with Relations Between Items

Klaus Rollmann Wendrey Cardoso Vinícius Lima Flávio Miyazawa

Abstract

This work deals with a variation of the 0-1 two-dimensional knapsack problem, where each item has a value and there is also a value between each pair of items. The value of adding one item into the packing depends not only on the item's value, but also on its relation values with other items that are in the packing. We adapted some inputs commonly found on the literature to our problem and we propose some exact formulations to solve the problem. Our formulations use integer programming models that work in two phases: the first finds a packing considering relaxed constraints, and the second checks if the packing is feasible. Three approaches to check the packing feasibility were compared. Moreover, we propose four decoders to the BRKGA meta-heuristic and compare their quality with the solutions found using exact models. Computational results are also provided to show the quality of our methods.

1 Introduction

The discussed problem on this work consists in the 0-1 knapsack problem in its two-dimensional version considering relations between items, in which each pair of items has a value if both items are packed together. This value could be either a benefit or a penalty. We call our problem two-dimensional knapsack problem with relations between items, and sometimes refer to it as simply 2D-KPRI.

This version of the knapsack problem has multiple applications in some real-world problems. It could be the case of storing items in a warehouse where you want to make a better use of the space to fit as many valuable items as possible but also you do not want to put together items that do not fit together as electronics and food, or medicine and poison. Another example is selecting which products should appear in some limited advertisement space, where you want to maximize the area covered and, at the same time, put similar items together.

Note that 2D-KPRI is a generalization of several other problems. If you remove all items values and set all relations to zero, then you end up with the two-dimensional orthogonal packing problem (2OPP), in which the only goal is to check if all items can be packed inside the container. There are several approaches to this problem, like Constraint Programming provided by Clautiaux [Clautiaux et al., 2008], branch-and-bound [Martello et al., 2000] and several heuristics proposed by [Hokama et al., 2016].

If you add value to items, but not relations between items, you get the classic 0-1 knapsack problem in its two-dimensional version. Another variation can be obtained if you

consider item values and add the relations between some items to be very negative, in this case you have the two-dimensional disjunctively constrained knapsack problem (2D-DCKP) [de Queiroz et al., 2017], in which some items must not be packed together.

2 The 2D Knapsack Problem with Relation Between Items

Let B be a rectangular container with width $W \in \mathbb{Z}_+^*$ and height $H \in \mathbb{Z}_+^*$. Let I be a set of items, such that each item $i \in I$ has width $w_i \in \mathbb{Z}_+^*$, height $h_i \in \mathbb{Z}_+^*$, value $v_i \in \mathbb{Z}_+$ and each pair of items $i \neq j$ have a relation value $r_{ij} \in \mathbb{Z}$. The problem is to find a packing P_I of the items I in the bin B that maximize the value V of the bin, where $V = \sum_{i \in P_I} v_i + \sum_{i,j \in P_I} r_{ij}$. That is, V is the sum of all packed items value plus the sum off all relations between any pair of packed items.

A packing P_I must satisfy the following packing constraints: (i) the packing must be orthogonal, that is, the edges of the items must be parallel to the container's edges; (ii) the packing must be oriented, that is, the items must be packed in the original given orientation; (iii) the items must be packed within the container's boundaries, that is, if the item i is packed with its left-bottom corner on the point (x_i,y_i) then $0 \leqslant x_i \leqslant x_i + w_i \leqslant W$ and $0 \leqslant y_i \leqslant y_i + h_i \leqslant H$; (iv) items must not overlap, that is, if the region occupied by item i is given by $R(i) = [x_i, x_i + w_i] \times [y_i, y_i + h_i]$ then $R(i) \cap R(j) = \emptyset$ for all pairs $i \neq j \in P_I$.

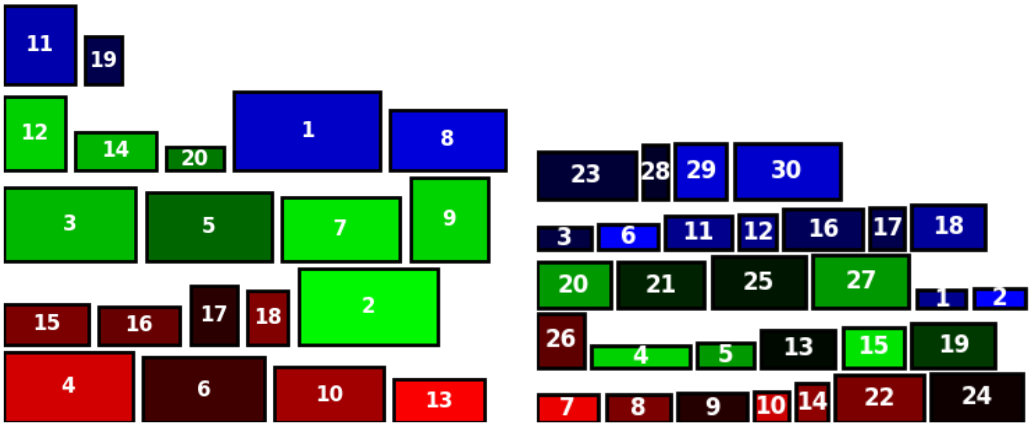


Figure 1: (a) All 20 possible items considered in input 3s (b) All 30 possible items considered in input STS2s

The Figure 2 describes two inputs for 2D-KPRI, in which there are three types of items: red, green and blue, and each item has a value represented by lighter or darker colors. Items with the same color have a positive benefit if added together and items of different colors have a penalty if added together.

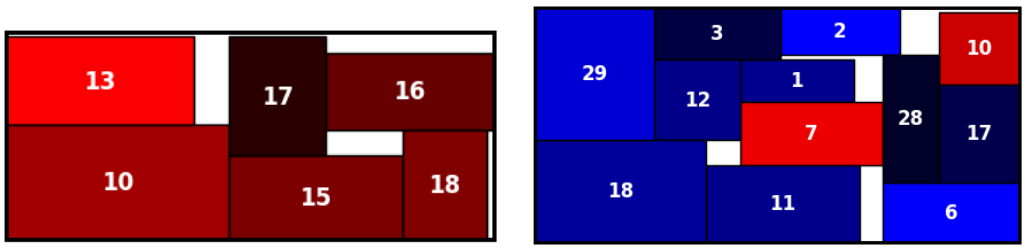


Figure 2: (a) Optimal packing for input 3s (b) Optimal packing for input STS2s

In Figure 2(a), we see that the packing prioritized items of the same color and avoided items of different colors. In Figure 2(b) we see that depending on the values of items and relations between them, it might be more beneficial to add a valuable item of different color even with a penalty associated.

Depending on the application, it is possible to adjust item values to be more or less important than relations between items.

3 Envelope and Corner Points

Some of the exact and heuristic algorithms that are proposed to solve the presented problem, use the concept of envelopes and corner points of a bin. Figure 3 shows an example to illustrate the following concepts.

Let I be a set of items, each item i with width w_i and height h_i. Let P_I a packing of those items in a bin of width W and height H. An envelope of P_I is then the region defined by $R(P_I) = \{(x, y) \in \mathbb{R}^2_+ : \exists\, i \in I,\, x < x_i + w_i,\, y < y_i + h_i\}$.

Let the complement of the envelope $\bar{R}(P_I)$ be the regions of the bin B that is not in the envelope $R(P_I)$. Then the corner points of the envelope P_I can be defined by $C(P_I) = \{(x, y) \in \bar{R}(P_I) : \nexists\, (x', y') \in \bar{R}(P_I) \setminus \{(x, y)\},\, x' \leqslant x,\, y' \leqslant y\}$.

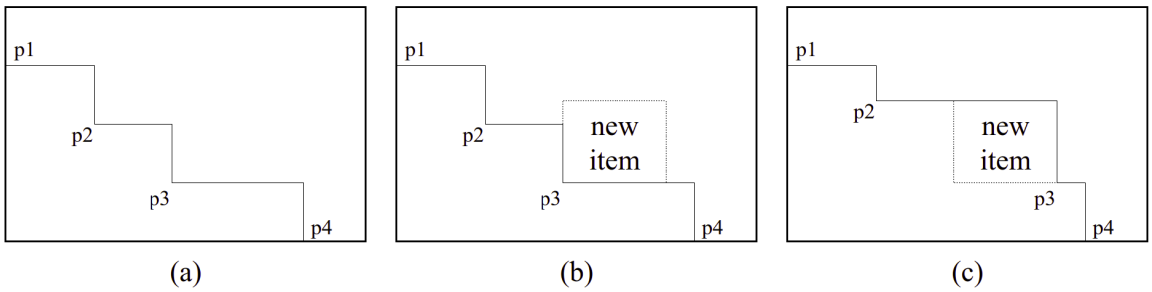


Figure 3: (a) An envelope and its corner points (b) New item added to the bin in point $p3$ (c) New envelope and its new corner points

4 Exact Method

4.1 Formulation

The definition of the problem described before can be formulated as an integer programming model. This initial formulation is referred as an 1-Phase formulation, in contrast to the 2-Phase that will be presented later. Here we adapted the formulation used in [Beasley, 1985].

4.1.1 Input

The input of the problem is as follows:

- n is the number of items available
- W, H are the width and height of the bin
- $P = \{0, 1, 2, ..., W-1\}$ are all possible x coordinates that an item can be placed
- $Q = \{0, 1, 2, ..., H-1\}$ are all possible y coordinates that an item can be placed
- I is the set of items, where $I = \{1, 2, 3, ..., n\}$
- $r_{i,j}$ is the relationship value between items i and j and indicates the benefit or penalty of putting items i and j in the bin
- w_i is the width of the item $i \in I$
- h_i is the height of the item $i \in I$
- v_i is the value of the item $i \in I$

4.1.2 Variables

The variables of the problem are:

- $x_{i,p,q} \in \{0, 1\}$. Variable that indicates if item i is placed at point $(p, q) \in P \times Q$.
  It is considered that an item i is placed at a position (p, q) if its bottom left corner is placed at (p, q)

- $y_{i,j} \in \{0, 1\}$. Variable that indicates if item i and item j were selected.

To simplify writing the constraints, it is also defined:

- The variables z_i that indicates if item $i \in I$ is packed in the solution.
  $z_i = \sum_{p \in P} \sum_{q \in Q} x_{i,p,q}, \quad \forall i \in I$
- The set $\xi_{p,q}$ that contains all possible items and positions such that, if the item is placed in this position, it covers the point (p, q).
  $\xi_{p,q} = \{(i, a, b) \mid i \in I, (a, b) \in P \times Q, a \leq p \leq a + w_i \wedge b \leq q \leq b + h_i\}$
  It is assumed that an item i covers a point (p, q) if it can be placed at a position (a, b) such that $(p, q) \in [a, a + w_i] \times [b, b + h_i]$.

4.1.3 Model

Given the variables and inputs, the objective is to maximize the sum of values and relationships between items that are packed in the bin, this can be written as:

$$\begin{aligned}
\text{maximize} \quad & \sum_{i=1}^{n}\sum_{j=i}^{n} y_{i,j} r_{i,j} + \sum_{i=1}^{n} z_i v_i \\
\text{subject to} \quad (i) \quad & \sum_{p,q \in P \times Q} x_{i,p,q} \leq 1, && \forall i \in I \\
& y_{i,j} \leq z_i, \\
(ii) \quad & y_{i,j} \leq z_j, && \forall i,j \in I, \quad i < j \\
& z_i + z_j - y_{i,j} \leq 1 \\
(iii) \quad & \sum_{(i,a,b) \in \xi_{p,q}} x_{i,a,b} \leq 1, && \forall (p,q) \in P \times Q.
\end{aligned}$$

The constraints are needed to avoid overlapping between the items and to avoid putting the same item multiple times in the solution. The first set of constraints imposes that an item may be placed only at one position. The set of constraints in (ii) ensures that variable $y_{i,j} = 1$ if and only if both items i and j are chosen and $y_{i,j} = 0$ otherwise. The last set of constraints is to avoid two items to overlap, or in other words, avoid two items to cover the same point (p, q).

4.2 Normal Patterns or Canonical Dissections

The number of integer variables, in special the variables $x_{i,p,q} \in \xi_{p,q}$, grows very fast if the dimensions of the container are increased. For each possible point inside the container there is a set $\xi_{p,q}$ with all variables that cover that point. All of these are integer variables.

In order to reduce considerably the number of variables, it is considered the computation of the normal pattern coordinates. The normal pattern coordinates were introduced by [Herz, 1972] (who called them canonical dissections) and by [Christofides and Whitlock, 1977].

The set of normal patterns is defined as the set of all possible width (or height) combinations that an item can be positioned, considering a solution in which all items are moved to the left and down until they touch another item. [Herz, 1972] shows that there is no loss of generality if we consider only solutions with the items moved to the left and down.

$$\mathcal{N} = \left\{ x = \sum_{j \in I} w_j \epsilon_j : 0 \leq x \leq W, \epsilon \in \{0,1\}, \forall j \in I \right\}.$$

For example, if we have a container of width $W = 15$ and three items with widths $\{4, 5, 7\}$, the set of normal patterns for P (horizontal axis) is:

$$P_0 = \{0, 4, 5, 7, 9, 11, 12\}.$$

We can see that using normal patterns we reduce the number of possible positions from 15 to 7, since no item can be packed at any other position considering the equivalent configuration in which all items are moved to the left and bottom.

The algorithm 1 is used to compute normal patterns for both directions vertical and horizontal.

Algorithm 1 Normal Patterns

```
T ← [0...W]: array initialized to 0
T[0] ← 1
for i ∈ I do
  for p = W − w_i to 0 do
    if T[p] = 1 then
      T[p + w_i] ← 1
    end if
  end for
end for
N ← ∅
for P = W to 0 do
  if T[p] = 1 then
    N ← N ∪ {p}
  end if
end for
return N
```

To reduce even more the set of possible points of the items, we can compute the normal patterns for each item separately [Boschetti et al., 2002]. First notice that if every combination of items that that generates a position p has the item i, then i may not be placed in p. In the example presented before, the item of size 4 cannot be placed in the position 9, because to generate the position 9 we used this item. Therefore, the normal pattern for an item i is defined as the normal pattern of all items excluding itself.

$$\mathcal{N}_i = \left\{ x = \sum_{j \in I \setminus \{i\}} w_j \epsilon_j : 0 \leq x \leq W, \epsilon \in \{0,1\}, \forall j \in I \setminus \{i\} \right\}. \quad (1)$$

4.3 2-Phase Algorithm

Solving the 2D-KPRI directly by the exact model presented in the Section 4.1.3 may take an impractical computational time, even if we reduce the set of points to the normal patterns. A better approach is to use a 2-phase algorithm. This approach is used in [de Queiroz et al., 2017] and it consists in using an Outer Method that gives a packing P_I, that contains items satisfying some relaxed constraint, and an Inner Method that checks if these items fit into the container B or not. If the items fit, then we know that we have the best solution, because the Outer Method provided the best solution with relaxed constraints. If, otherwise, the items do not fit, we add a new constraint prohibiting the outer

method from choosing the same solution again. The next time we call the outer method, it will avoid this solution and get the next one. This process is repeated until we find a solution. This algorithm is described in Algorithm 2

Algorithm 2 2-Phase Algorithm

while time limit T is not reached or solution is not found **do**
  Get packing P_I by solving the Outer Method
  Check if the packing P_I is feasible using the Inner Method
  if P_I is a feasible packing **then**
    return P_I
  else
    Add a new constraint to Outer Method prohibiting P_I
  end if
end while

Given that the bottleneck is to check if P_I is a valid packing, we can change Algorithm 2 slightly and provide a heuristic. The heuristic consists in limiting the amount of time given to the inner method. The intuition is that if the inner method is taking too much time to check if the packing is feasible, then probably it is unfeasible. Algorithm 3 shows this heuristic.

Algorithm 3 2-Phase Heuristic

while timelimit T is not reached or solution is not found **do**
  Get packing P_I by solving the Outer Method
  Run the Inner Method with timelimit T' to check the packing P_I
  if the Inner Method takes more than T' time or returns infeasible **then**
    Add a new constraint to Outer Method prohibiting P_I
  else
    return P_I
  end if
end while

4.4 Relaxation for the Outer Method

We used a slight variation of the one-dimensional bin packing as the outer method. Here, we maximize the sum of item values and relations considering a relaxed constraint in which only the area of the items are considered.

The model below was used for the outer method:

$$\begin{aligned}
\text{maximize} \quad & \sum_{i=1}^{n}\sum_{j=i}^{n} y_{i,j}p_{i,j} + \sum_{i=1}^{n} z_i v_i \\
\text{subject to} \quad & y_{i,j} \leq z_i, \\
& y_{i,j} \leq z_j, \\
& z_i + z_j - y_{i,j} \leq 1 \qquad \forall\{(i,j) \in I^2 \mid i \leq j\} \\
& \sum_{i=1}^{n} z_i w_i h_i \leq W \times H
\end{aligned}$$

We can see that this problem is a relaxation of our original problem and it does not take into account overlaps between items, only its area. The packing that this problem provides is an upper bound solution for the 2D-KPRI.

4.5 Single Bin Packing Check

The inner method of the 2-Phase Algorithm checks if the items that were selected by the outer method, which provides an upper bound solution, is a valid set of items that can be packed inside the container. If there is a way to pack the items inside the container, then the solution of the relaxed problem is also a solution for our original problem.

Three approaches are used to verify if the items fit inside the container: Branch and Bound, Constraint Programming and Integer Linear Programming.

4.5.1 Branch and Bound

This approach uses a branch and bound algorithm to place items into possible envelope positions, cutting the branch if the remaining area is smaller than the area of the remaining items. The code we used is a simplification of the algorithm OneBin described in [Martello et al., 2000], which is for 3D orthogonal packing problems and is available at the author's website [http://www.diku.dk/˜pisinger/]

The code was simplified in the sense that it was adapted to consider only two dimensions instead of three.

4.5.2 Constraint Programming

The Constraint Programming method is described in [Clautiaux et al., 2008] and uses only variables and constraints in order to find a valid solution. The variables are X_i and Y_i and they represent the x and y coordinates that an item i is positioned.

The constraints used are to avoid two items to overlap, or to avoid an item to cross the container boundaries. These constraints are formulated as boolean *or* expressions.

To avoid overlap, there is a constraint for each pair of items $i, j \in P_I$:

$$(X_i + w_i \leq X_j) \vee (X_j + w_j \leq X_i) \vee (Y_i + h_i \leq Y_j) \vee (Y_j + h_j \leq Y_i).$$

The second constraint is to keep an item inside the container's borders. For each item $i \in P_I$:

$$(X_i + w_i \leq W) \wedge (Y_i + h_i \leq H).$$

We also add a set of possible values for each variable. Here we use the normal patterns provided in Eq. 1 and limit the possible values of X_i and Y_i:

$$X_i \in \mathcal{N}_i^W,$$

$$Y_i \in \mathcal{N}_i^H.$$

Here $\mathcal{N}^W$ implies that the normal patterns are computed for the horizontal axis and $\mathcal{N}^H$ for the vertical axis.

4.5.3 Integer Linear Programming

The integer linear programming method is very similar to the 1-Phase algorithm, but it checks only the feasibility of the overlapping constraints and forces the solution to contain all items. The model is described below

$$\begin{aligned} &\text{maximize} && 1 \\ &\text{subject to} && (i) \sum_{p,q \in P \times Q} x_{i,p,q} = 1, \quad \forall i \in I \\ &&& (ii) \sum_{(i,a,b) \in \xi_{p,q}} x_{i,a,b} \leq 1, \quad \forall (p,q) \in P \times Q. \end{aligned}$$

Constraint (i) is to ensure that all items are added to the packing. The second constraint is the same as the one described in the 1-Phase exact model.

4.6 Reducing variables in the ILP Inner Method

This section provides the description of two variations that reduce the set of variables of the ILP Inner Method. For every possible point (p, q) and item i, it can be calculated the area that will be wasted if we put the item in this position. In figure 4.6(a) it is shown the four regions that are affected by placing the item i at position (p, q). If we consider these regions as 1-dimensional bins - two horizontal of sizes p and $W - p - w_i$ and two vertical of sizes q and $H - q - h_i$ - we can calculate the maximum filling of these bins using the remaining items and get a lower bound for the waste generated.

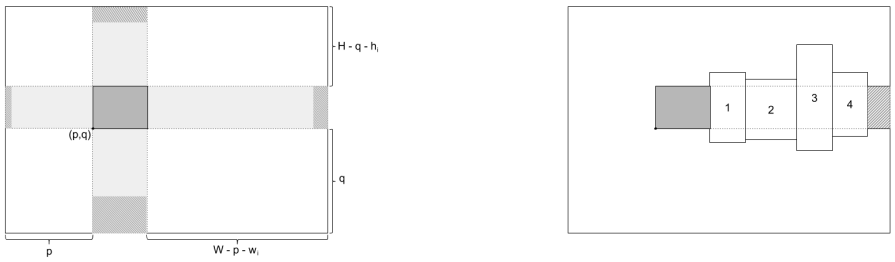


Figure 4: (a) Wastes when item i is at position (p, q) (b) Example of waste calculated using the result of the 1D knapsack problem for the horizontal bin at the right side.

The figure 4.6(b) shows an example considering one of the bins in which the best possible way to fill the bin is using w_1, w_2, w_3 and w_4. It is easy to see that the area wasted is a lower bound to the area that will be wasted if we put the item i in position (p, q) when all the items are packed. Now, if the remaining area minus the sum of the wastes provided by all four bins is smaller that the area of the remaining items then we know that there is no possible solution with item i at position (p, q), and we can remove this point from the set of possible points for item i.

4.6.1 Knapsack with 4 bins

To calculate the waste provided by the 4 bins, we solve the knapsack problem using 4 bins considering each item with all other items. This can be solved using a generalization of the dynamic programming algorithm for the 0-1 knapsack problem in which we fill a 4D array instead of 1D array. For each item i and point (p, q) we will have different bin sizes, thus we would have to solve the 0-1 knapsack problem with 4 bins several times. But, all of those solutions are subproblems of the problem of solving the 0-1 knapsack with 4 bins of sizes W, W, H and H. For that reason, we only have to compute it once per item and keep the 4D array (of size $W \times W \times H \times H$) associated to that item. Also, it is not necessary to compute all W and H positions, only those in $\mathcal{N}^W$ and $\mathcal{N}^H$. This reduces the memory consumption to one 4D array of size $|\mathcal{N}^W| \times |\mathcal{N}^W| \times |\mathcal{N}^H| \times |\mathcal{N}^H|$ for each item in the packing.

4.6.2 Knapsack with 2 bins

The amount of memory and time consumed by solving the knapsack with 4 bins might be intractable. We tried another method using 2 bins instead of 4 in order to reduce the 4D array to a 2D array. The bins used are for horizontal and vertical directions, and this allows an item to be repeated in the two horizontal bins or in the two vertical bins. This reduces

the memory consumption to $|P_I| \times |\mathcal{N}^W| \times |\mathcal{N}^H|$ but might provide a smaller lower bound for the waste.

5 Heuristic Method

5.1 A Brief Introduction to BRKGA

A biased random-key genetic algorithm (BRKGA), as proposed by [Gonçalves and Resende, 2011] and which API was developed by [Toso and Resende, 2011], was used as a general search meta heuristic for finding near-optimal solutions to the proposed problem. A BRKGA has three key features that specialize genetic algorithms:

- A chromosome encoding using a sequence of random keys or genes in the interval [0,1)
- A process to evolve a set of chromosomes, *i.e.* the population
- The introduction of new mutants chromosomes as individuals in the population

The main task of the meta heuristic is to use the chromosome to construct a solution to the underlying knapsack problem from which the objective function value or fitness can be computed.

The algorithm has the following parameters:

- n_c as the size of the chromosome or number of chromosome's genes
- p as the number of chromosome (or individuals) in a population
- p_e as the fraction of a population to be considered as elite individuals
- p_m as the fraction of a population to be replaced for new mutants
- ρ_e as the probability of a chromosome's gene to be inherited from elite individuals
- k as the number of independent populations
- ex_g as the number of generations before exchanging elite individuals among populations
- ex_n as the number of elite individuals to be exchanged
- t as the algorithm maximum running time

The BRKGA starts with k populations of p individuals, each having n_c genes. These populations are evolved until the algorithm met the stop criteria of running for t seconds. After every ex_g generations a ex_n number of individuals are exchanged among the k populations in order to increase convergence of the algorithm.

In a given current generation, each chromosome is decoded to construct a solution and has its fitness calculated. The next generation will consist of (i) the p_e elite individuals with the best fitness values, (ii) the p_m random generated chromosomes and (iii) $p - p_e - p_m$ evolved chromosomes that are produced by mating two chromosomes of the current generation selected at random.

5.2 BRKGA for the Knapsack Problem

For the knapsack problem, a chromosome of size equal to the number of items can be used to generate a random sequence of items index as a potential packing sequence. These items are then picked and if possible, packed in the bin accordingly to a few packing heuristics: Strip, Bottom-Left, Bottom-Left or Left-Bottom and Minimum Waste Corner. When the bin is filled and the decodification is finished, we have the chromosome mapped into a feasible solution and a fitness value.

In the executed tests, the following parameters values were used:

Variable	Meaning	Value
n_c	size of chromosome	n or 2n
p	size of population	1000
p_e	fraction of elite individuals	0.20
p_m	fraction of mutants individuals	0.10
ρ_e	probability to get elite gene	0.70
k	independent population	3
ex_g	generations until exchange	100
ex_n	individuals to be exchanged	2
t	algorithm running time (s)	900

5.3 Strip Decoder

The packing sequence is given by the chromosome of size n, where n is the total number of items available. Then, the objective is to try to pack the items set I in the bin B.

In a given time, the Strip Decoder method picks the i-th item is the given sequence and, if feasible, that is, if restrictions $x_i + w_i \leqslant W$ and $y_i + h_i \leqslant H$ are satisfied, puts the item to the right of i-1-th item, that is, pack the item i into position (x_i, y_i), such that $x_i = x_{i-1} + w_{i-1}$ and $y_i = y_{i-1}$.

If not feasible, then the item i overloads the bin at that position. If the constraint $y_i + h_i \leqslant H$ is not satisfied, then the item i is left unpacked and the i+1-th item is picked next. If the constraint $x_i + w_i \leqslant W$ is not satisfied, consider $y_i = max(y_k + h_k : k \in P_I)$, that is, let y_i be equal to the highest point of all packed items so far, then the decoder checks if it is feasible to pack the item i into position $(0, y_i)$. If it is still not feasible, the item i is left unpacked and the next item is picked.

Algorithm 4 $Pack(P_I, i, x, y)$

```
if x + w_i ⩽ W and y + h_i ⩽ H then
    x_i ← x
    y_i ← y
    P_I ← P_I ∪ i
    return true
end if
return false
```

Algorithm 5 $getBenefit(P_I)$

$benefit \leftarrow 0$
for item $i \in P_I$ **do**
  $benefit \leftarrow benefit + v_i$
end for
for all pair of item $i, j \in P_I : i \neq j$ **do**
  $benefit \leftarrow benefit + r_{i,j}$
end for
return $benefit$

Algorithm 6 $StripDecoder(chromosome)$

$x \leftarrow 0$, $y \leftarrow 0$, $height \leftarrow 0$
$P_I \leftarrow \emptyset$
$solution \leftarrow getPermutation(chromosome)$
for item i in $solution$ **do**
  if $pack(P_I, i, x, y)$ **then**
    $x \leftarrow x + w_i$
    if $height < y + h_i$ **then**
      $height \leftarrow y + h_i$
    end if
  else if $pack(P_I, i, 0, heigth)$ **then**
    $x \leftarrow w_i$
    $y \leftarrow height$
  end if
end for
return $getBenefit(P_I)$

5.4 Bottom-Left Decoder

For this decoder, the packing sequence is given by the chromosome of size n, where n is the total number of items available. Considering a partial packing P_I and a set of corner points $C(P_I)$, the packing position of a item i in the bin B is given by the eligible bottom-left corner point $p = (x_p, y_p) \in C(P_I)$, such that $x_p + w_i \leqslant W$, $y_p + h_i \leqslant H$ and y_p is minimum. If an eligible corner cannot be found, then the item i is discarded from the packing.

Algorithm 7 $BottomLeftDecoder(chromosome)$

```
corner_points ← (0, 0)
P_I ← ∅
solution ← getPermutation(chromosome)
for item i in solution do
    p ← getBottomLeftCornerPoint(i, corner_points)
    pack(P_I, i, p_x, p_y)
    updateCornerPoints(i, p, corner_points)
end for
return getBenefit(P_I)
```

5.5 Bottom-Left or Left-Bottom Decoder

Given a chromosome M of size $2n$ and a set I of n items and $0 \leqslant k \leqslant n$, then M_{2k} are related to the items indexes while M_{2k+1} related to the items positioning, which could be chosen by a bottom-left or by a left-bottom approach. Let L be a threshold value, then if $M_{2k+1} < L$ the decoder will try to pack item M_{2k} following the bottom-left corner point heuristic, otherwise will try to pack following left-bottom corner point heuristic.

Algorithm 8 $LeftBottomLeftDecoder(chromosome)$

```
corner_points ← (0, 0)
P_I ← ∅
solution ← getPermutation(chromosome)
pos ← getPosition(chromosome)
for item i in solution do
    if pos_i < L then
        p ← getBottomLeftCornerPoint(i, corner_points)
    else
        p ← getLeftBottomCornerPoint(i, corner_points)
    end if
    pack(P_I, i, p_x, p_y)
    updateCornerPoints(i, p, corner_points)
end for
return getBenefit(P_I)
```

5.6 Minimum Waste Corner Decoder

The packing sequence is given by the chromosome of size n, where n is the total number of items available. For each item i, this decoder's heuristic packs the item in the corner point that generates the minimum waste and do not violate any constraints by exceeding the bin's limit. The decoder calculates the waste generated by packing the item i in all feasible corner points and chooses the one that generates the minimum waste.

Algorithm 9 $MinWasteDecoder(chromosome)$

```
corner_points ← (0, 0)
P_I ← ∅
solution ← getPermutation(chromosome)
for item i in solution do
    p ← getMinWasteCornerPoint(i, corner_points)
    pack(P_I, i, p_x, p_y)
    updateCornerPoints(i, p, corner_points)
end for
return getBenefit(P_I)
```

6 Computational Experiments

The proposed algorithms were run over a set of 32 instances. We considered 13 instances proposed by [Cintra et al., 2008] and 19 instances proposed by [Hifi and Roucairol, 2001]. The inputs were modified to have the number of items, the bin's dimensions, the items' dimensions and values, and all the relations between any two items. While the number of items and items' and bin's dimensions were used as from the original instances, the values of the items and their relations were generated randomly. In Figure 2 we show two examples of our input.

To generate the inputs we assigned a color (red, blue or green) to the each item and then assigned a random intensity between 0-255. Then we assign the value of the relation between two items as the absolute difference between their intensities, and set positive for items of the same color and negative for items of different colors. Then we tried to balance the importance of an item value and its relation values. To do that, we estimate the number of items that will be packed, dividing the container's area by the average area of items, and assign each item's value to be its intensity times the estimate of the number of items packed.

All algorithms were implemented in C++ language. The integer linear programming models were solved using the Gurobi 7.5.2 and the constraint programming model was solved using the CPLEX Studio 12.7.1. The tests were run in a Ubuntu 16.10 and a Intel(R) Core(TM) i7-2600 processor with 8 cores, 3.4 GHz and 8 GB of RAM. A time limit of 3600 seconds was imposed for each input for the exact methods and 600 seconds for each input for the meta-heuristic methods.

In Table 1 we mark the inputs in which the optimal solution was found with '*' and we highlight the first method to find the solution in bold. It is possible to see that the 2-Phase

Table 1: Comparing different 2-Phase Exact Methods

Input	Exact 2P BNB		Exact 2P CP		Exact 2P ILP	
	Value	Time	Value	Time	Value	Time
2s*	**20552**	**0.00**	20552	0.03	20552	0.11
3s*	**5951**	**1.23**	5951	21.14	5951	2.42
A1s*	**6345**	**2.77**	6345	22.69	6345	17.58
A2s*	**7913**	**3.14**	7913	174.51	7913	64.09
A3*	**14448**	**5.85**	-	3600.00	14448	2170.39
CHL1	-	3600.00	-	3600.00	-	3600.00
CHL2*	**12751**	**0.01**	12751	1.66	12751	1.31
CW1*	**14139**	**51.85**	-	3600.00	-	3600.00
CW2*	**8817**	**364.80**	-	3600.00	8817	2819.48
CW3*	**11439**	**43.01**	-	3600.00	11439	883.05
HH*	**15420**	**0.00**	15420	0.02	15420	0.03
Hchl2	-	3600.00	-	3600.00	-	3600.00
Hchl3s*	**31584**	**0.00**	31584	0.06	31584	1.60
Hchl6s*	**31506**	**440.00**	-	3600.00	-	3600.00
Hchl7s	-	3600.00	-	3600.00	-	3600.00
Hchl8s*	**27979**	**0.00**	27979	0.04	27979	0.09
Hchl9	-	3600.00	-	3600.00	-	3600.00
STS2s*	**21211**	**20.18**	-	3600.00	-	3600.00
STS4s*	**36931**	**0.30**	-	3600.00	-	3600.00
gcut10dr*	**4580**	**17.66**	4580	20.28	4580	18.89
gcut11dr*	**7195**	**60.02**	7195	403.92	7195	114.27
gcut12dr*	**6595**	**665.05**	6595	779.94	6595	773.82
gcut13dr	-	3600.00	-	3600.00	-	3600.00
gcut1dr*	**3086**	**0.34**	3086	0.58	3086	0.43
gcut2dr*	**5984**	**9.22**	5984	10.00	5984	11.07
gcut3dr*	**7153**	**36.62**	7153	55.95	7153	65.80
gcut4dr*	**8990**	**119.55**	8990	744.41	8990	391.13
gcut5dr*	**5008**	**0.28**	5008	0.54	5008	0.72
gcut6dr*	**4276**	**22.79**	4276	24.47	4276	24.10
gcut7dr*	**4474**	**305.19**	4474	324.79	4474	318.44
gcut8dr*	**8956**	**34.46**	8956	1799.41	8956	112.02
gcut9dr*	**6236**	**0.10**	6236	2.00	6236	0.46

Table 2: Comparing Average Percentage of Points removed in 2-Phase ILP with improvements described in section 4.6

Input	Exact 2P ILP	Exact 2P ILP w/ 4 bins		Exact 2P ILP w/ 2 bins	
	Time	% variables removed	Time	% variables removed	Time
2s	0.11	0.00%	0.18	0.00%	1.02
3s	2.42	47.17%	1.98	51.58%	2.75
A1s	17.58	49.04%	15.60	53.16%	17.52
A2s	64.09	76.26%	29.23	78.89%	22.84
CHL2	1.31	19.96%	0.92	30.71%	0.67
HH	0.03	0.00%	0.02	0.00%	0.16
Hchl3s	1.60	0.00%	1.61	0.00%	11.51
Hchl8s	0.09	0.00%	0.09	0.00%	0.42
gcut10dr	18.89	76.59%	18.18	80.27%	19.61
gcut1dr	0.43	57.09%	0.44	62.24%	0.64
gcut2dr	11.07	73.94%	9.90	79.17%	10.74
gcut5dr	0.72	55.35%	0.41	60.44%	0.66
gcut6dr	24.10	84.86%	23.17	86.48%	24.30
gcut9dr	0.46	33.49%	0.26	34.80%	0.49

algorithm reached the optimal solution for 27 out of 32 inputs and that using Branch and Bound in the second-phase was superior, for all inputs solved.

Table 2 compares the two improvements described in section 4.6. The improved methods were executed using a time limit of 100s. Only the inputs that were solved by both 2-Phase ILP methods in under 100s are shown. To show the influence of the improvements we calculate the percentage of variables removed for each P_I provided by the outer method and then we take the mean of those percentages for all P_I and show the results . The percentage of points removed can be seen as the percentage of variables $x_{i,p,q}$ that are removed from the inner ILP model on average. The results show that there is not a significant difference between using all 4 bins when computing the 1-dimensional knapsack problem and using only 2 bins. Both improvements removed around 19 - 86% of variables in dense packings, in which the sum of the area of the items in the packing is close to the area of the container and 0% in packings that are sparse, where the area of the items in the packing is much less than the area of the container. We show that in general, using 4 bins leads to a slower method, which indicates that finding a solution for the knapsack problem with 4 bins becomes time and memory consuming. The use of only 2 bins reduces less variables, but is very fast and consumes much less memory, and at the end provided good results.

Table 3: Solutions found with heuristic 2-Phase BNB using 1h and 600s

Input	Upperbound	Value Found	%
Hchl2	52192	51963	99.56%
Hchl9	27312	27027	98.96%

The heuristic proposed in section 4.3 was tested using several different combinations of time limits. Using 1 hour total time and 600 seconds for the inner BNB method, we obtained the same optimal solutions for all 27 out of 32 inputs in less or equal time, and we were able to find upper bound solutions to 2 of the remaining 5. In table 3 we show that the solution found is very close to the upperbound. Using a total time of 600s and with 30s for the inner BNB method, we found solutions in less time for the simpler inputs, but in general there was not much improvement. Using 100s as total time limit and 2s in the inner BNB method we were able to find solutions close to optimal in a short time, but on average worse than those provided by the meta-heuristic in the same configuration. In short, the 2-Phase branch and bound heuristic is good if you want to provide a good solution for a small set of inputs and can spend some time adjusting the time limits.

Of the four developed algorithms for BRKGA, the Minimum Waste Decoder and the Bottom-Left or Left-Bottom Decoder had the best convergence time in some previous tests, where all the heuristics algorithms run for only 1 second. For all instances, one or both of these two were able to get the best bin value between all the meta-heuristic methods. All the convergence graphs for the chosen instances are available in the Attachments section.

It was decided to run these two best BRKGA algorithms for 10 minutes in order to better analyze and to compare them with the exact methods. Table 4 shows the value of the best solution found and the time that the solution is found for the BRKGA's Left-Bottom or Bottom-Left and Minimum Waste. In the table, the value found by the heuristic method that correspond to the optimal solution are marked with '*' and the lowest running time between the heuristics is in bold. The instances for which no optimal solution was found are marked with '+' and the exact results shown are related to the upperbound, while for all others instances the 2P BNB column shows the optimal solution data.

For a better comparison with the exact methods, the Table 5 shows the percentage of the obtained valued in comparison with the optimal value, or the upperbound when no optimal solution is found, and execution time of the heuristic in comparison with the 2P BNB exact method.

Table 6 shows the average comparison between the heuristic methods and the 2P BNB exact method. For the average values calculated on this table, only the instances in which the exact model found the optimal solution were considered.

Table 6: Heuristic Method Results

Method	LeftBottomLeft	MinWaste
Optimal Solutions (of 27)	18	17
Average Best Value	99.40%	99.05%
Average Running Time	64.23%	61.52%

7 Conclusion

The 2-Phase exact model could solve 27 out of 32 inputs (approximately 84.4%) to optimality using a timelimit of 3600s. Using Branch and Bound in the second phase of the algorithm outperformed the other methods that use constraint programming and integer

Table 4: Comparing different BRKGA Heuristics Methods

Input	Exact 2P BNB		Heuristic LeftBottomLeft		Heuristic MinWaste	
	Value	Time	Value	Time	Value	Time
2s	20552	0.00	**20552***	**0.01**	**20552***	**0.01**
3s	5951	1.23	5951*	0.06	**5951***	**0.04**
A1s	6345	2.77	**6345***	**2.21**	6276	0.09
A2s	7913	3.14	**7913***	**0.14**	7913*	07.06
A3	14448	5.85	**14201**	**7.86**	14122	0.26
CHL1+	36852+	3600.00+	35044	134.64	**35788**	**0.79**
CHL2	12751	0.01	12751*	0.02	**12751***	**0.01**
CW1	14139	51.85	14128	151.42	**14139***	**308.15**
CW2	8817	364.80	**8663**	**0.76**	8203	0.15
CW3	11439	43.01	11439*	0.38	**11439***	**0.29**
HH	15420	0.00	**15420***	**0.00**	**15420***	**0.00**
Hchl2+	52192+	3600.00+	**50213**	**64.92**	50053	1.15
Hchl3s	31584	0.00	**31365**	**0.01**	30921	0.01
Hchl6s	31506	440.00	30958	0.35	**30958**	**0.26**
Hchl7s+	50748+	3600.00+	48014	279.19	**48844**	**0.89**
Hchl8s	27979	0.00	**27979***	**0.01**	27689	0.03
Hchl9+	27312+	3600.00+	**26442**	**1.77**	26442	9.50
STS2s	21211	20.18	20837	0.48	**20837**	**0.47**
STS4s	36931	0.30	35154	0.45	**36096**	**0.21**
gcut10dr	4580	17.66	**4580***	**0.06**	4580*	0.08
gcut11dr	7195	60.02	7195*	0.26	**7195***	**0.16**
gcut12dr	6595	665.05	6550	0.63	**6550**	**0.34**
gcut13dr+	82864+	3600.00+	78530	3.84	**79584**	**3.37**
gcut1dr	3086	0.34	**3086***	**0.01**	**3086***	**0.01**
gcut2dr	5984	9.22	5984*	40.19	**5984***	**2.69**
gcut3dr	7153	36.62	**7153***	**0.06**	7153*	0.09
gcut4dr	8990	119.55	**8733**	**0.67**	8467	0.26
gcut5dr	5008	0.28	**5008***	**0.01**	**5008***	**0.01**
gcut6dr	4276	22.79	4276*	0.04	**4276***	**0.02**
gcut7dr	4474	305.19	4474*	0.13	**4474***	**0.06**
gcut8dr	8956	34.46	8956*	0.28	**8956***	**0.26**
gcut9dr	6236	0.10	**6236***	**0.02**	**6236***	**0.02**

Table 5: Comparing Heuristics Methods with Exact 2P BNB

Input	Heuristic LeftBottomLeft		Heuristic MinWaste	
	% Value	% Time	% Value	% Time
2s	100.00%	100.00%	100.00%	100.00%
3s	100.00%	4.88%	100.00%	3.25%
A1s	100.00%	79.78%	98.91%	3.25%
A2s	100.00%	4.46%	100.00%	224.84%
A3	98.29%	134.36%	97.74%	4.44%
CHL1+	95.09%	-	97.11%	-
CHL2	100.00%	200.00%	100.00%	100.00%
CW1	99.92%	292.03%	100.00%	594.31%
CW2	98.25%	0.21%	93.04%	0.04%
CW3	100.00%	0.88%	100.00%	0.67%
HH	100.00%	100.00%	100.00%	100.00%
Hchl2+	96.21%	-	95.90%	-
Hchl3s	99.31%	100.00%	97.90%	100.00%
Hchl6s	98.26%	0.08%	98.26%	0.06%
Hchl7s+	94.61%	-	96.25%	-
Hchl8s	100.00%	100.00%	98.96%	300%
Hchl9+	96.81%	-	96.81%	-
STS2s	98.24%	2.38%	98.24%	2.33%
STS4s	95.19%	150.00%	97.74%	70.00%
gcut10dr	100.00%	0.34%	100.00%	0.45%
gcut11dr	100.00%	0.43%	100.00%	0.27%
gcut12dr	99.32%	0.09%	99.32%	0.05%
gcut13dr+	94.77%	-	96.04%	-
gcut1dr	100.00%	2.94%	100.00%	2.94%
gcut2dr	100.00%	435.90%	100.00%	29.18%
gcut3dr	100.00%	0.16%	100.00%	0.25%
gcut4dr	97.14%	0.56%	94.18%	0.22%
gcut5dr	100.00%	3.57%	100.00%	3.57%
gcut6dr	100.00%	0.18%	100.00%	0.09%
gcut7dr	100.00%	0.04%	100.00%	0.02%
gcut8dr	100.00%	0.81%	100.00%	0.75%
gcut9dr	100.00%	20.00%	100.00%	20.00%

linear programming. The idea of improving the inner ILP method by reducing variables that are not possible given the waste that it generates showed good improvement and we were able to improve the integer linear programming method. A further work might include similar approaches adapted to the method that uses branch and bound.

As shown in the Table 4 and in the images on the Attachments section, the BKRGA heuristics have a fast convergence, meaning that within a small running time these methods can obtain a solution that is near or even the optimal solution. In almost every instance we can observe that at least one of the heuristics could solve the problem and obtain the optimal solution. Although the algorithms were run for 600 seconds per input, most of the best values were found within the first seconds of execution.

In the Table 5 we can observe that in the worst case scenario the Left-Bottom or Bottom-Left Heuristic got a answer that is 98.24% of the optimal value (for instance *STS2s*), while the Minimum Waste Heuristic got a worst of 94.18% of the optimal value (for instance *gcutdr4*). In most of the cases, both BRKGAs could find the optimal solution with a running time significantly smaller (less than 5%) than the exact 2P BNB algorithm. Even when the 2P BNB exact method could not found the optimal solution, the BRKGA heuristics were able to get a value that is around 95% the upperbound given by the exact method.

In the Table 6 we have a compilation of the heuristic methods. The Left-Bottom or Bottom-Left Method found the optimal solution for 18 instances out of 27 and has the average best value of 99.40%, being the best in this criteria, with an average running time of 64.23% of the 2P BNB exact method. The Minimum Waste Heuristic was able to get the optimal solution for 17 out of 27 instances, although it has the best average running time of 61.52% the 2P BNB exact method, it stays slightly behind on the average best value with 99.05% of the optimal solution, yet a good average solution.

Hence, we can conclude that the two BRKGA methods, using a Bottom-Left or Left-Bottom Decoder or a Minimum Waste Decoder, can achieve a good solution in a practical running time, while it was able to find the optimal solution for many of the used instances.

References

[Beasley, 1985] Beasley, J. (1985). An exact two-dimensional non-guillotine cutting tree search procedure. *Operations Research*, 33(1):49–64.

[Boschetti et al., 2002] Boschetti, M. A., Mingozzi, A., and Hadjiconstantinou, E. (2002). New upper bounds for the two-dimensional orthogonal non-guillotine cutting stock problem. *IMA Journal of Management Mathematics*, 13(2):95–119.

[Christofides and Whitlock, 1977] Christofides, N. and Whitlock, C. (1977). An algorithm for two-dimensional cutting problems. *Operations Research*, 25(1):30–44.

[Cintra et al., 2008] Cintra, G., Miyazawa, F., Wakabayashi, Y., and Xavier, E. (2008). Algorithms for two-dimensional cutting stock and strip packing problems using dynamic programming and column generation. *European Journal of Operational Research*, 191:61–85.

[Clautiaux et al., 2008] Clautiaux, F., Jouglet, A., Carlier, J., and Moukrim, A. (2008). A new constraint programming approach for the orthogonal packing problem. 35:944–959.

[de Queiroz et al., 2017] de Queiroz, T. A., Hokama, P. H. D. B., Schouery, R. C. S., and Miyazawa, F. K. (2017). Two-dimensional disjunctively constrained knapsack problem: Heuristic and exact approaches. *Computers & Industrial Engineering*, 105(Supplement C):313 – 328.

[Gonçalves and Resende, 2011] Gonçalves, J. F. and Resende, M. G. C. (2011). Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics*, 17:487–525.

[Herz, 1972] Herz, J. C. (1972). Recursive computational procedure for two-dimensional stock cutting. *IBM Journal of Research and Development*, 16(5):462–469.

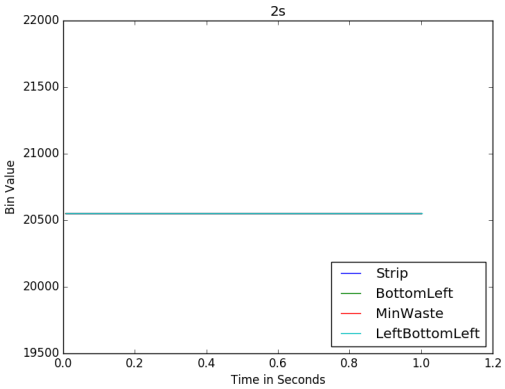
[Hifi and Roucairol, 2001] Hifi, M. and Roucairol, C. (2001). Approximate and exact algorithms for constrained (un) weighted two-dimensional two-staged cutting stock problems. *Journal of Combinatorial Optimization*, 5:465–494.

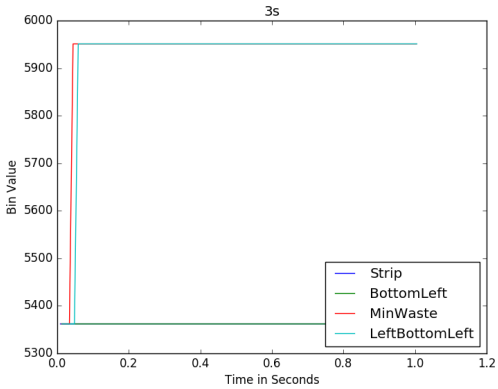
[Hokama et al., 2016] Hokama, P., Miyazawa, F. K., and Xavier, E. C. (2016). A branch-and-cut approach for the vehicle routing problem with loading constraints. *Expert Systems with Applications*, 47:1–13.

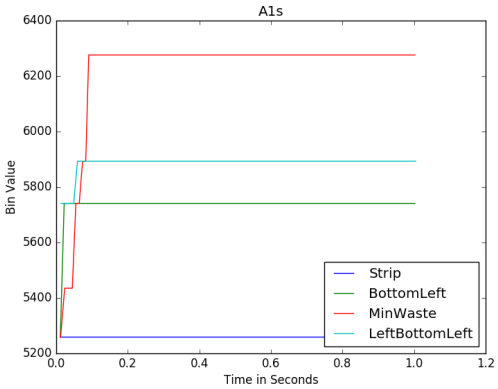
[Martello et al., 2000] Martello, S., Pisinger, D., and Vigo, D. (2000). The three-dimensional bin packing problem. *Operations Research*, 48(2):256–267.

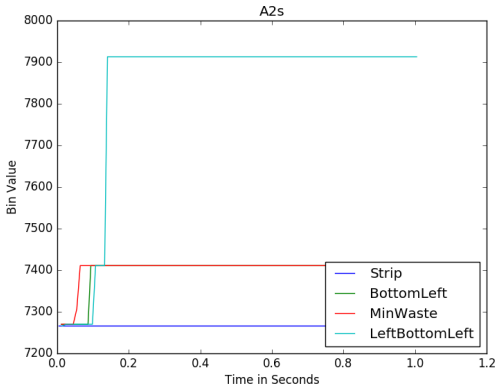
[Toso and Resende, 2011] Toso, R. F. and Resende, M. G. C. (2011). A c++ application programming interface for biased random-key genetic algorithms.

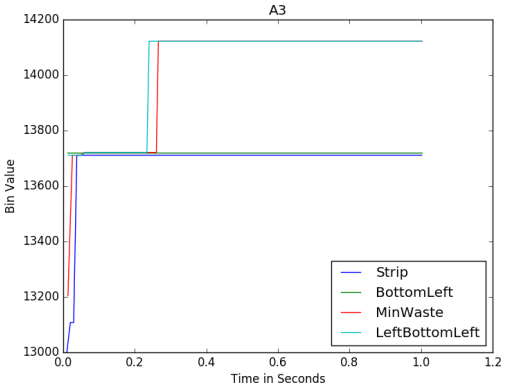
8 Attachments: Heuristic Methods Convergence Graphs

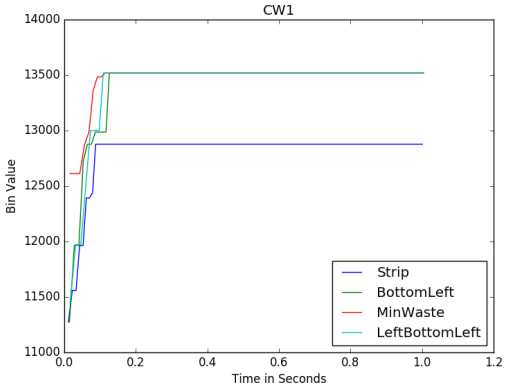


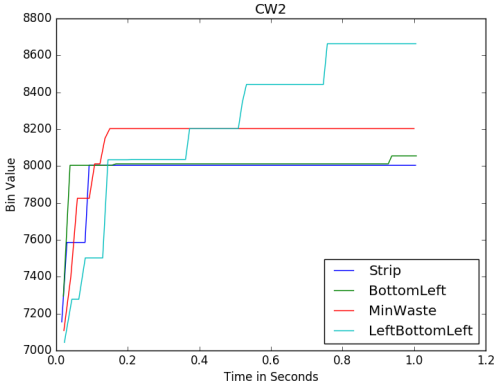


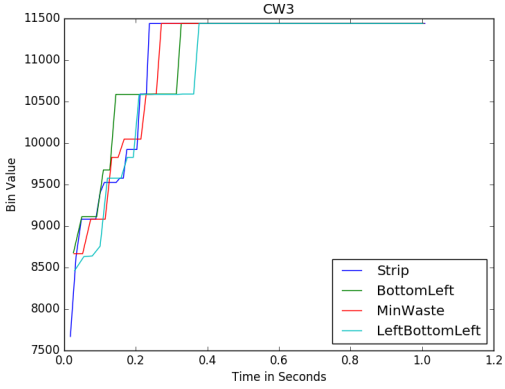


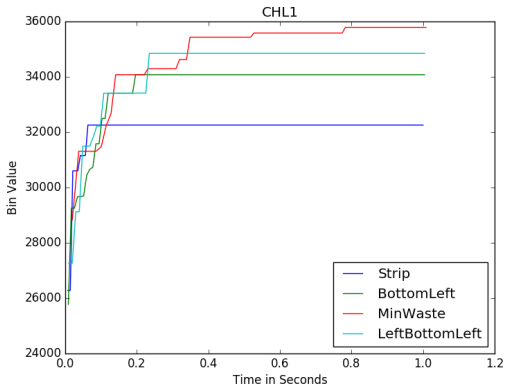


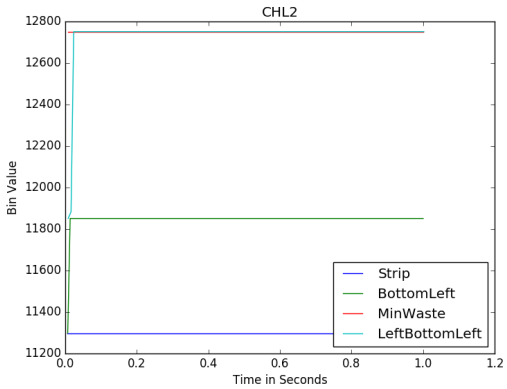


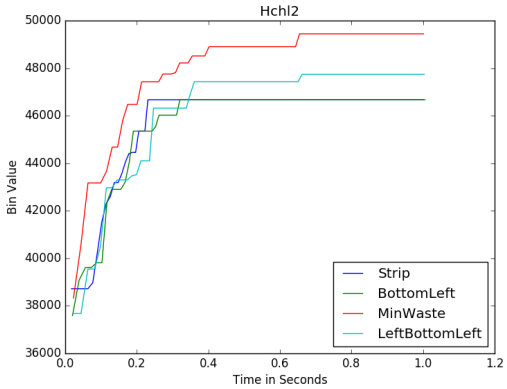


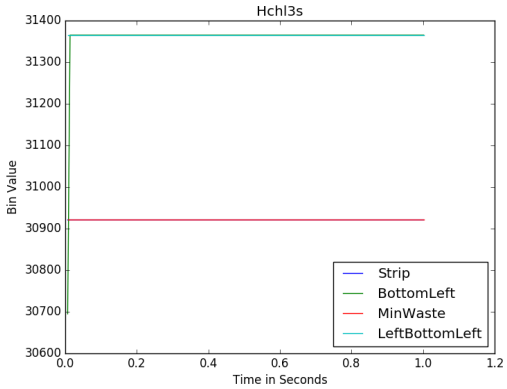


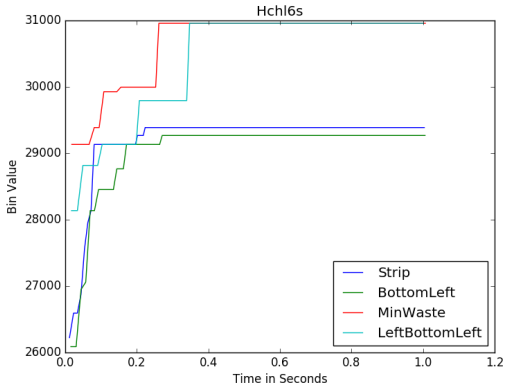


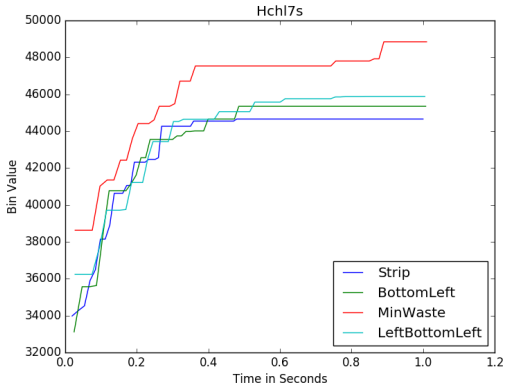

CHL1
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft

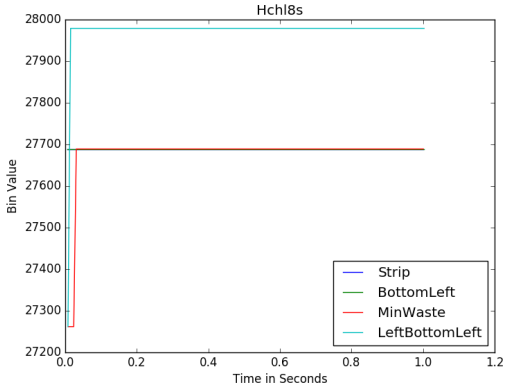

CHL2
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft

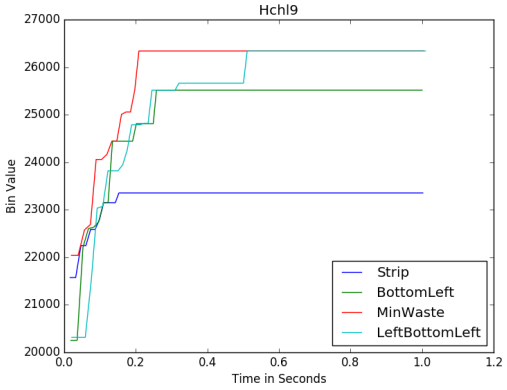

Hchl2
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft

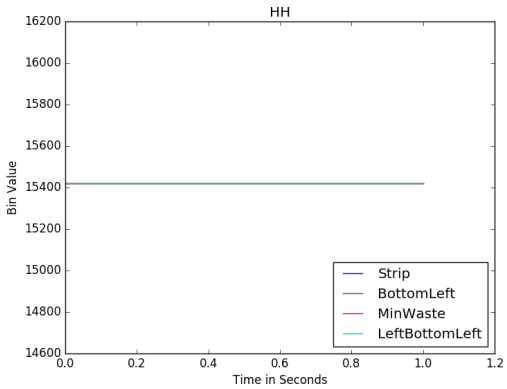

Hchl3s
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft

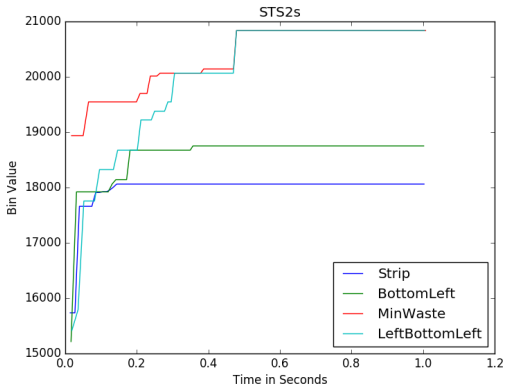

Hchl6s
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft

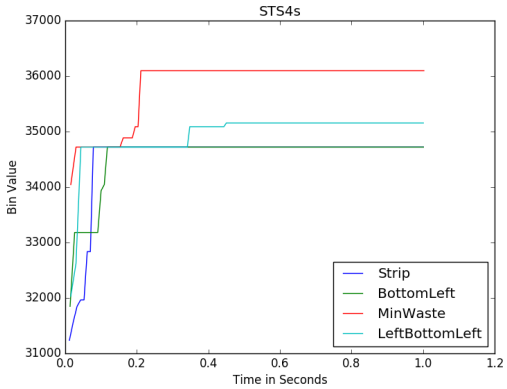

Hchl7s
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft

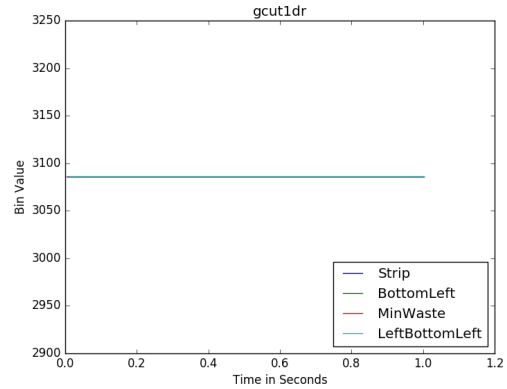

Hchl8s
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft

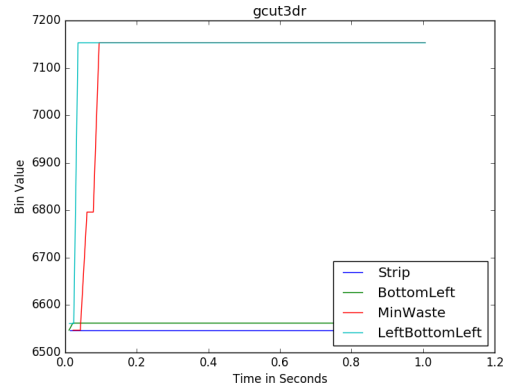

Hchl9
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft

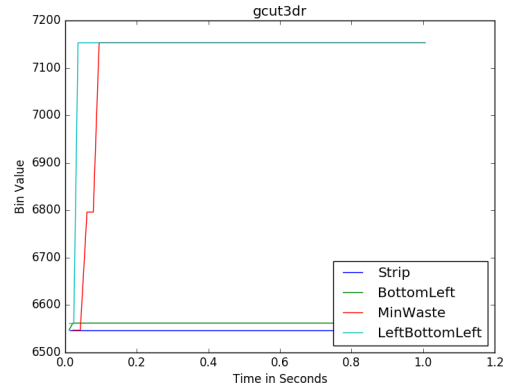

HH
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft

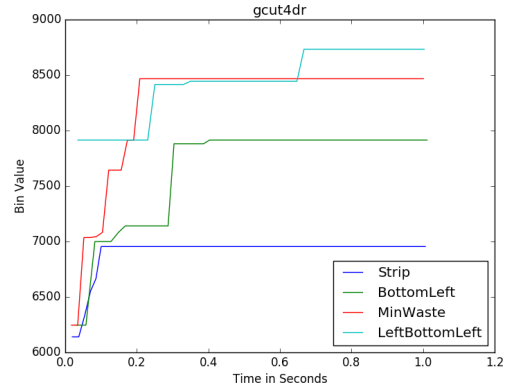

STS2s
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft

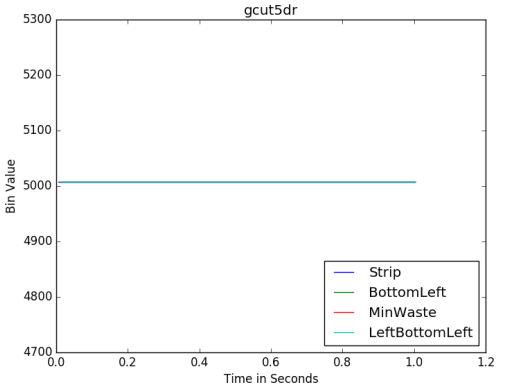

STS4s
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft

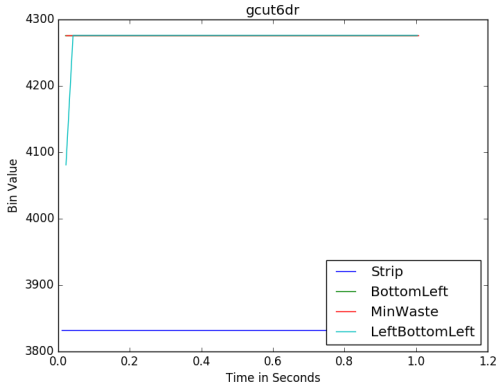

gcut1dr
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft

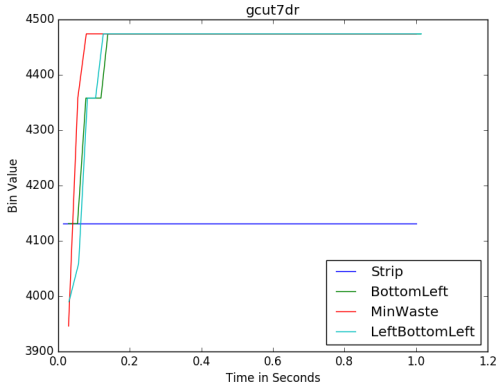

gcut3dr
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft

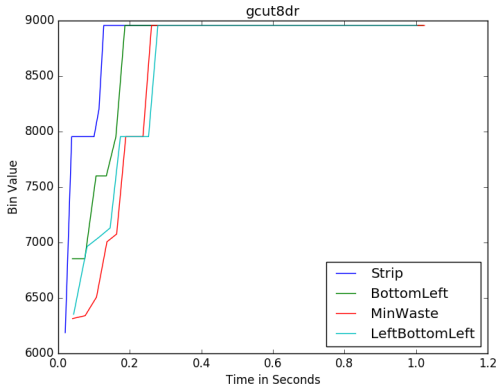

gcut3dr
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft

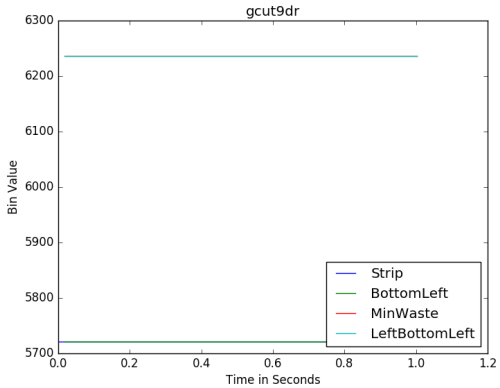

gcut4dr
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft

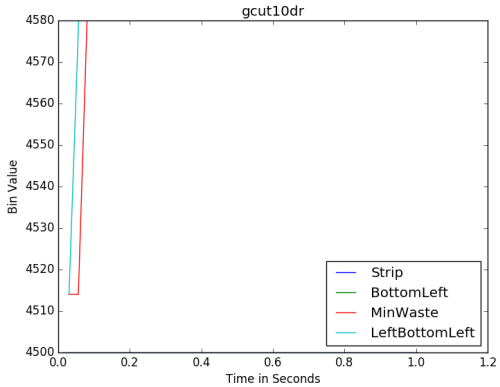

gcut5dr
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft

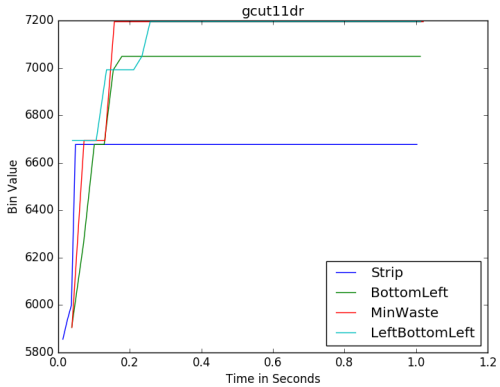

gcut6dr
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft

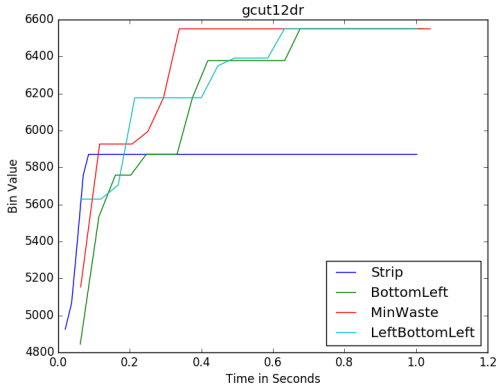

gcut7dr
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft

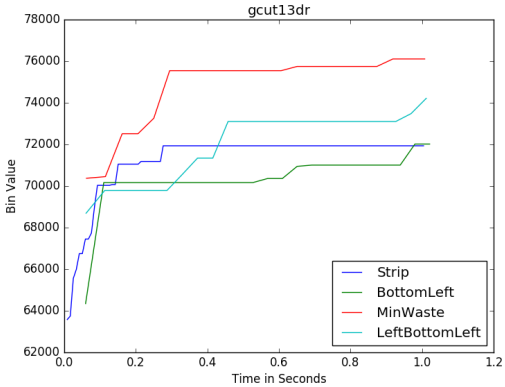

gcut8dr
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft


gcut9dr
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft


gcut10dr
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft


gcut11dr
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft


gcut12dr
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft


gcut13dr
Bin Value
Time in Seconds
Strip
BottomLeft
MinWaste
LeftBottomLeft