U.Porto Journal of Engineering, 1:1 (2015) 35-49
ISSN 2183-6493
DOI 10.24840/2183-6493_001.001_0005

Received: 3 January, 2015
Accepted: 12 February, 2015
Published: 1 October, 2015

Rectangular Bin-Packing Problem: a computational evaluation of 4 heuristics algorithms

Duarte Nuno Gonçalves Ferreira[1]
[1]Department of Industrial Engineering and Management, Faculty of Engineering University of Porto, Porto, Portugal (dferreira@fe.up.pt)

Abstract
The Rectangular Bin-packing Problem, also known as The Two-dimensional Bin-packing Problem (2DBPP), is a well-known combinatorial optimization problem which is the problem of orthogonally packing a given set of rectangles into a minimum number of two-dimensional rectangular bins. In this article we benchmark four heuristics: constructive, based on a First Fit Decreasing strategy, local search using a greedy packing First-Fit algorithm, Simulated Annealing with multiple cooling values and Genetic Algorithm. All implementations are written in Python, run using the Pypy environment and the new multiprocessing module. All implementations were tested using the Berkey and Wang and Martelo and Vigo Benchmark Instances.

Subject Headings. Optimization, computer programming, operational research.
Author Keywords. Genetic Algorithm, Simulated Annealing, Bin packing problems.

1. Introduction

The Rectangular Bin-packing Problem, also known as The Two-dimensional Bin-packing Problem (2DBPP) is a well-known combinatorial optimization problem. The 2DBPP is the problem of orthogonally packing a given set of rectangles into a minimum number of two-dimensional rectangular bins (Pisinger and Sigurd 2007). It is a Cutting and Packing (C&P) problem and belongs to the NP-hard type of problems (see (Michael and David 1979)). Cutting and Packing problems are a recurring subject for articles in optimization literature because of its wide applications in solving real-world problems. This created a need for a typology to emerge for C&P problems. Dyckhoff (1990) suggested a typology in 1990 and in 2007, Wäscher, Haußner and Schumann (2007)proposed an improved typology of C&P problems which is the one currently used.

From a conceptual point of view, cutting and packing are equivalent terminologies though researchers often differentiate between bin packing problems and cutting stock problems along the line so that for a few equal items, the problem is considered a bin packing problem, while for many equal items, the problem is considered a cutting stock problem (Wäscher, Haußner, and Schumann 2007). One common aspect in C&P problems is that, typically, small items have to be packed into, or cut from, one or more large objects (often called bins). There are in the literature many variations of the basic packing problem, it can be one, two, three or N-dimensional in nature and the aim might be to maximize the value of the items packed or minimizing the wasted space. The objects can be heterogeneous, have a regular or irregular shape and the container or plate for cutting can be limited or infinite.

Exists in the literature some variations, and respective solutions, for the 2DBPP (Berkey and Wang 1987; Hong et al. 2014; Lodi, Martello, and Vigo 2002). This variations are based on constraints:

- We can set a fixed maximum number of bins or set a group of different sized bins (Alvarez-Valdés, Parreño, and Tamarit 2013; Wei et al. 2013);
- We can limit the orientation of the items or enable a 90º rotation(Blum and Schmid 2013; Sarabian and Lee 2010);
- We can force a guillotine cut[1];
- We can impose a cost on each item (Pisinger and Sigurd 2005).
- Lodi et al. (1999)introduce four possible BPP subtypes based on the orientation and guillotine cut constraints:
- 2DBPP|R|F: Items may be rotated by 90º (R), guillotine cut constraint not imposed (F);
- 2DBPP|R|G: Items may be rotated by 90º (R), guillotine cut constraint is imposed (G);
- 2DBPP|O|F: Orientation of items is fixed (O), guillotine cut constraint not imposed (F);
- 2DBPP|O|G: Orientation of items is fixed (O), guillotine cut constraint is imposed (G).

In this article, on section 2, we will start by introducing the type of two-dimensional bin packing problem we are going to study. On section 3 we will present the algorithms we are going to implement and on section 4 we will present the benchmark instances that are used throughout the literature related to this problem. On section 5 we will discuss our results, comparing the number of bins and execution times of all algorithms. On section 6 we will do a brief conclusion of the work and present some future work.

2. Problem Description

In this article we are going to study the Rectangular Bin-packing Problem as stated in the following description (Pisinger and Sigurd 2007):

Assume that a set $\Re$ = {1,..., n} of rectangles is given, rectangle ı having width wı, and height hı. We can use an infinite number of bins to pack the items, each bin has a width W and a height H. The objective is to minimize the number of bins used to pack all rectangles in $\Re$ such that they do not overlap.

Following the Lodi et al. (1999) subtypes we are going to focus on the 2DBPP|O|F, fixed orientation and no guillotine cut.

For the sake of simplicity from now on, when we refer to 2DBPP in the article, we are referring to this definition.

3. Solution Approaches

The implementations done in this study follows each heuristic and meta-heuristic implementation basic ideas from the description of the papers in the literature(Bays 1977; Kirkpatrick, Gelatt, and Vecchi 1983; Osogami and Okano 2003; Wei et al. 2013). Nevertheless these implementations are our own work and will most likely give different results concerning execution time and efficiency, from the ones already found in the literature, due to using different programming and testing platforms. The implementations are going to be described in the subsection of each heuristic and meta-heuristic. For this article we are going to test the implementation of four heuristics to solve the 2DBPP problem: a constructive heuristic, a local search, a simulated annealing and a Genetic algorithm meta-heuristic. The solutions are represented by the list of items that was used as input to the packing algorithm. Also all meta-heuristics used the same packing algorithm, based on a First-Fit Strategy.

[1] A guillotine cut splits a block into two smaller blocks, where the slice plane is parallel to one side of the initial block.

3.1. Tables

Our constructive heuristic is based on the First-Fit-Decreasing (FFD) Heuristic(Bays 1977). As we can see on the Algorithm 1, first the list of pieces is ordered in a non-increasing order and each piece is set on the first bin it fits. In our implementation we only visit each bin once. We run through the list of pieces and try to fit each piece in the current bin if the piece fits, we look for a new piece in the remaining list, to fit in the remaining free space. If there is no piece on the list that fits, we close the current bin and open a new one, going back to the start of the piece's list and repeating the process.

```
Require: Piece list is sorted by non-increasing order
Ensure: binlist = ∅
    While listpieces != ∅ do
        For piece ∈ listpieces do
            if fitsIn(piece,bin) then
                addTo(piece,bin)
                removeFrom(piece, listpieces)
            End if
        End for
        addTo(bin,binlist) bin = newBin()
    End while
```

***Algorithm* 1:** First Fit Decreasing

3.2. Local Search

Our Local Search meta-heuristic(Osogami and Okano 2003) which can be seen in the Algorithm 2, starts with an initial solution since we do not have a limit in the number of bins and we can state that each piece fits completely inside one bin. As we will see in the section 4, any list of pieces is a feasible solution, the worst case being having one bin for each piece. The possible initial solution is then tested using a greedy algorithm to position the pieces inside the bins, a Fit-First packing method. After, we test the neighborhood of the initial solution for a better solution. If we find a better neighbor, we use it as a starting point for a new neighborhood and restart the search for a better neighbor until we cannot find a better neighbor.

The neighborhood is defined by switching two random pieces position with each other and a better neighbor is one that can be packed in a smaller number of bins. A better neighborhood can also have the same number of bins but the bin that is less filled has a smaller occupied area than the less filled bin of the current solution. If the less filled bin has a smaller occupied area than the other, the remaining bins are better packed and we are probably closer to a solution with less bins.

```
$x_n$ is the initial solution
$x* = xn$
$F^* = Packing\ (x_n)$
$improvecount = 2 * len(x_n)$
While $improvecount > 0$ & $time < 60$ do
    $x_{n+1} = SendRandomToEnd(x_n)$
    If $Packing\ (x_n) > Packing\ (x_{x+1})$ then
        $x^* = x_n+1$
        $F^* = Packing\ (x_{n+1})$
        $xn = xn+1$
        $improvecount = 2 * len(x_n)$
    Else
        $improvecount - = 1$
    End if
End while
```

Algorithm 2: Local Search

3.3. Simulated Annealing

The Simulated Annealing(Kirkpatrick, Gelatt, and Vecchi 1983) implementation is the second meta-heuristics we are going to implement. It consists of a random global search for a better solution, enabling the selection of a temporary worst solution in the quest to find a better one. The description of the algorithm can be seen on the Algorithm 3.

Using, as a metaphor, a pan of boiling water, initially the selection of a next solution is chaotic and so, we can select almost any neighbor to be the center of our new neighborhood, being it better or worse than the one we have. As time goes by and we iterate through the outer while loop, we readjust the temperature applying a freezing rate (α) to reduce the chaos in the selection of the new neighborhood center and narrow our selection. Eventually the temperature would be so low that the environment would be static and the water would freeze.

When we reach the limit of iterations without change, or in our case, we reach the execution time limit, we stop the algorithm.

The neighborhood implementation is the same as with the local search: we swap two pieces with each other.

```
$x_n$ is the initial solution
$x* = xn$
$F^* = Packing\ (x_n)$
$T_n = initialTemperature\ ()$
$improvecount = 5 * len\ (x_n)$
$L_n = 2 * len(x_n)$
**While** $improvecount > 0$ & $time < 60$ **do**
    **For** $k = 1$ to $L_n$ **do**
        $x = randomNeighbour(x_n)$
        **If** $Packing\ (x_n) >= Packing(x)$ **then**
            $x_n = x$
            **If** $F^* > Packing(x)$ **then**
                $x^* = x$
                $F^* = Packing(x)$
            **End if** $improvecount = 2 * len(x_n)$
        **Else**
            **If** $radom(0,1) <= p(x_n)$ **then**
                $x_n = x$
                $improvecount = 2 * len(x_n)$
            **Else**
                $improvecount- = 1$
            **End if**
        **End if**
    **End for**
    $update\ (L_n, T_n)$
**End while**
```

Algorithm 3: Simulated Annealing

3.4. Genetic Algorithm

The Genetic Algorithm is a meta-heuristic based on the theory of evolution of species by Darwin(Chu and Beasley 1998). The base of the algorithm is that we have a population of solutions, in this context called chromosomes, which can evolve into better solutions by using the same processes that affect every chromosome inside every organism in nature, survival of the fittest, crossovers and mutations. Through this evolutionary process, the population evolves towards an optimum solution. The evolutionary process works through three methods: survival of the fittest; mutations; and crossover of chromosomes.

Survival of the fittest means that the better solution in a population has a higher probability of survival so it has a higher chance of being in the next generation.

Crossover is a process of reproduction where two chromosomes swap part of their components with each other and create two new chromosomes, each with parts from both parents. The idea is that by joining two already good solutions we can create a better one. However, with only this process the population would evolve into a homogeneous population. This is a bad thing because homogeneous populations are very weak against changes in the environment. Mutations can help keep the population more heterogeneous. Mutations create changes in a chromosome, in our case, this can create chromosomes that are different from the ones already in the population and expand the population with new solutions, better or worse than the current ones. Thus, this is a mechanism to escape local optima.

```
$g$ is the total of generations we will test $t = 0$
$P_t$ is the Population at time t (generation)
$x^*$ is the best solution
initialize ($P_t$)
evaluate ($P_t$)
$x^* = BestPacking (P_t)$
While $t < g$ do
    $t = t + 1$
    $P_t = evolve (P_{t-1})$ evaluate ($P_t$)
    $x_n = BestPacking(P_t)$
    If $x_n < x^*$ then
        $x* = xn$
    End if
End while
```

Algorithm 4: Genetic Algorithm

4. Benchmark Instances

For this 2DBPP problem there are two benchmark instances or data set generation techniques that appear the most in literature: Berkey and Wang (1987); and Martello and Vigo(1998). Pisinger & Sigurd(2005)created some benchmark instances based on this two, Hopper & Turton(2001) generated their own benchmark instances.

Berkey and Wang Benchmark Instances are divided in 6 classes:

- Class 1 W = H = 10, $h_ı$ and $w_ı$ uniformly random in [1, 10];
- Class 2 W = H = 30, $h_ı$ and $w_ı$ uniformly random in [1, 10];
- Class 3 W = H = 40, $h_ı$ and $w_ı$ uniformly random in [1, 35];
- Class 4 W = H = 100, $h_ı$ and $w_ı$ uniformly random in [1, 35];
- Class 5 W = H = 100, $h_ı$ and $w_ı$ uniformly random in [1, 100];
- Class 6 W = H = 300, $h_ı$ and $w_ı$ uniformly random in [1, 100].

With W and H being the width and the height of the bins and $w_ı$ and $h_ı$ the width and height of a piece ı. The datasets are composed of 5 instances with n pieces (n = 20, 40, 60, 80,100)(Lodi, Martello, and Monaci 2002).

Martello and Vigo Benchmark Instances are divided in 4 classes which are filled with items from 4 types:

- Type 1 $h_ı$ uniformly random in [1, H/2], $w_ı$ uniformly random in [2W/3, W];
- Type 2 $h_ı$ uniformly random in [2H/3, H], $w_ı$ uniformly random in [1, W/2];
- Type 3 $h_ı$ uniformly random in [H/2, H], $w_ı$ uniformly random in [W/2, W];
- Type 4 $h_ı$ uniformly random in [1, H/2], $w_ı$ uniformly random in [1, W/2].

As before, W and H are the width and the height of the bins and wı and hı the width and height of a piece ı. For each item of an instance of Class k (k ∈ {I, II, III, IV }) the probability of an item being of type k is 70%, and 10% of being of any other type(Martello and Vigo 1998). These datasets are also composed of 5 instances with n pieces (n = 20, 40, 60, 80, 100) and W = H = 100 (Lodi et al., 2002b).

5. Computational Results

For our study we implemented the four algorithms and tested them against the benchmark instances from section 4. The tests were made in an Intel Core i7-3517U CPU with 4GB of ram, using Pypy 32bits runtime[2] in Windows 8.1. The runs had a soft limit of 60 seconds, so all instances could be run in a sensible time frame. The FFD never hit the time limit but all the others end up reaching the time limit in some of the instances. Without this time limit the results would have been better but this limit gave enough time to be able to compare the performance of each heuristic in a feasible time frame.

To take into account the overhead of the JIT[3], the Class 1 should have been run for all algorithms 3 times before running the full experiment and collecting the results. This would enable the JIT to compile the source code before collecting data. As it was implemented, running the instances of Class 1 took more time than it should, at least for the FFD implementation.

The results were computed and compared to one representation of each heuristic and meta-heuristic, the results are shown in Table 1 and Table 2. In those tables we compare the FFD, the Local Search with the starting solution in a decreasing order(LSearchD), a Simulated annealing implementation with an α of 95% and a Genetic algorithm with random initial population. Looking at tables 3, 4, 7 and 10 we can compare the differences in execution time between the implementations and is very clear the difference in the results.

The LSearchD execution time greatly increases with the number of pieces as opposed to the FFD where we see a slower increase. This difference comes from the fact that in LSearchD we are basically packing every solution in the neighborhood until we find a better neighbor whereas in the FFD we are only packing one list of items. A similar rule applies to the simulated annealing and genetic algorithm implementations where we also pack more than one solution. We do not see a higher impact because of the soft limit on the running time and number of generations.

Another thing we can clearly see in the results is that the LSearchD results are always the same or better than the FFD ones, which is due to fact that the packing algorithm and the initial solution order are the same. So, the initial solution for the LSearchD is the solution for the FFD with the same instance of the problem.

The results of the simulated annealing should have been the same or better than the Local Search, and in most cases are. In the cases where that is not the case, for example, Class 3 instance 80 or Class 5 instance 100, if we increase the soft limit we will see at least a solution as good as the one in local search.

The Genetic algorithm results are also generally better than the local search yet they are most of the time worse than the simulated annealing. An increase in the number of generations could improve these results.

[2] http://pypy.org/ - PyPy is a fast, compliant alternative implementation of the Python language (2.7.8 and 3.2.5).

[3] Just-in-time compilation (JIT), also known as dynamic translation, is compilation done during execution of a program at run time rather than prior to execution.

Class	N.Pieces	FDD		LSearchD		SA95		Genetic	
		Avg Bins	Avg Waste	Avg Bins	Avg Waste	Avg Bins	Avg Waste	Avg Bins	Avg Waste
	20	7.20	141.3	7.17	138.3	7.10	131.3	7.10	131.3
	40	13.60	199.9	13.60	199.9	13.41	180.9	13.65	204.9
1	60	20.30	237.1	20.10	217.1	20.09	216.1	20.56	263.1
	80	27.80	299.3	27.52	271.3	27.50	269.3	28.35	354.3
	100	32.50	252.4	32.30	232.4	32.04	206.4	33.32	334.4
	20	1.10	411.3	1.01	330.3	1.00	321.3	1.00	321.3
	40	2.00	639.9	2.00	639.9	2.00	639.9	2.00	639.9
2	60	2.90	817.1	2.80	727.1	2.66	601.1	2.78	709.1
	80	3.50	669.3	3.36	543.3	3.31	498.3	3.40	579.3
	100	4.20	782.4	4.09	683.4	4.14	728.4	4.20	782.4
	20	5.40	2548.2	5.34	2452.2	5.10	2068.2	5.27	2340.2
	40	10.00	3667.1	9.81	3363.1	9.61	3043.1	9.82	3379.1
3	60	14.70	4278.1	14.31	3654.1	14.38	3766.1	14.87	4550.1
	80	19.70	4851.9	19.63	4739.9	19.68	4819.9	20.66	6387.9
	100	23.60	5624.2	23.16	4920.2	23.41	5320.2	24.34	6808.2
	20	1.20	5908.2	1.00	3908.2	1.00	3908.2	1.00	3908.2
	40	2.00	7667.1	2.01	7767.1	1.97	7367.1	1.99	7567.1
4	60	2.90	9758.1	2.81	8858.1	2.79	8658.1	2.80	8758.1
	80	3.70	10331.9	3.43	7631.9	3.41	7431.9	3.40	7331.9
	100	4.20	9864.2	4.20	9864.2	4.20	9864.2	4.20	9864.2
	20	6.80	20169.5	6.59	18069.5	6.51	17269.5	6.55	17669.5
	40	12.30	26075.6	12.23	25375.6	12.17	24775.6	12.27	25775.6
5	60	18.70	35581.8	18.48	33381.8	18.59	34481.8	18.76	36181.8
	80	25.30	43284.2	25.00	40284.2	25.20	42284.2	25.82	48484.2
	100	29.40	41064.8	29.05	37564.8	29.22	39264.8	30.31	50164.8
	20	1.00	42169.5	1.00	42169.5	1.00	42169.5	1.00	42169.5
	40	1.90	74075.6	1.92	75875.6	1.90	74075.6	1.90	74075.6
6	60	2.80	100581.8	2.57	79881.8	2.67	88881.8	2.48	71781.8
	80	3.20	78284.2	3.12	71084.2	3.20	78284.2	3.16	74684.2
	100	4.20	125064.8	3.85	93564.8	4.20	125064.8	3.80	89064.8

Table 1: Comparing the number of bins and waste in each algorithm. Class 1-6

Class	N.Pieces	FDD		LSearchD		SA95		Genetic	
		Avg Bins	Avg Waste	Avg Bins	Avg Waste	Avg Bins	Avg Waste	Avg Bins	Avg Waste
	20	6.00	17603.0	5.78	15403.0	5.50	12603.0	5.51	12703.0
	40	11.70	25606.0	11.49	23506.0	11.48	23406.0	11.51	23706.0
7	60	16.20	27932.6	16.22	28132.6	16.12	27132.6	16.21	28032.6
	80	23.70	43518.4	23.39	40418.4	23.44	40918.4	23.63	42818.4
	100	27.70	43569.0	27.52	41769.0	27.53	41869.0	27.99	46469.0
	20	6.10	17617.4	5.98	16417.4	5.80	14617.4	5.82	14817.4
	40	11.80	25901.8	11.56	23501.8	11.49	22801.8	11.52	23101.8
8	60	16.80	31955.9	16.67	30655.9	16.53	29255.9	16.63	30255.9
	80	22.80	37124.2	22.72	36324.2	22.78	36924.2	22.94	38524.2
	100	28.60	49807.5	28.19	45707.5	28.28	46607.5	28.40	47807.5
	20	14.40	54918.7	14.30	53918.7	14.30	53918.7	14.30	53918.7
	40	28.00	104341.0	27.84	102741.0	27.80	102341.0	27.80	102341.0
9	60	43.90	167188.6	43.80	166188.6	43.70	165188.6	43.70	165188.6
	80	57.80	212771.2	57.74	212171.2	57.70	211771.2	57.70	211771.2
	100	69.60	250725.7	69.51	249825.7	69.50	249725.7	69.50	249725.7
	20	4.40	11274.7	4.37	10974.7	4.33	10574.7	4.38	11074.7
	40	7.70	13784.3	7.59	12684.3	7.64	13184.3	7.80	14784.3
10	60	10.80	18085.2	10.55	15585.2	10.54	15485.2	10.76	17685.2
	80	13.50	17828.5	13.21	14928.5	13.36	16428.5	13.83	21128.5
	100	16.40	16952.9	16.34	16352.9	16.39	16852.9	17.22	25152.9

Table 2: Comparing the number of bins and waste in each algorithm. Class 7-10

5.1. Constructive Heuristic

The constructive heuristic is fast. When comparing the constructive heuristic with the meta-heuristics used in this study, the FFD is blazing fast. Our implementation of the FFD ranges from just under nine milliseconds on a Class 1 instance with 60 pieces to 5 seconds in a Class 6 instance with 100 pieces, with most of the runs taking less than 100 milliseconds. Our implementation is faster in runs where we have big boxes when compared to the size of the pieces, as seen in Class 2, Class 4 and Class 10.

Class	N. Pieces	Avg. Bins	Avg. Waste	Avg. Time
1	20	7.20	141.3	15.17
	40	13.60	199.9	34.41
	60	20.30	237.1	8.73
	80	27.80	299.3	20.61
	100	32.50	252.4	10.67
2	20	1.10	411.3	24.02
	40	2.00	639.9	24.16
	60	2.90	817.1	19.01
	80	3.50	669.3	15.95
	100	4.20	782.4	17.88
3	20	5.40	2548.2	51.65
	40	10.00	3667.1	22.72
	60	14.70	4278.1	18.61
	80	19.70	4851.9	24.74
	100	23.60	5624.2	36.41
4	20	1.20	5908.2	34.93
	40	2.00	7667.1	43.77
	60	2.90	9758.1	96.87
	80	3.70	10331.9	157.93
	100	4.20	9864.2	191.73
5	20	6.80	20169.5	17.86
	40	12.30	26075.6	44.87
	60	18.70	35581.8	56.27
	80	25.30	43284.2	102.78
	100	29.40	41064.8	135.66
6	20	1.00	42169.5	608.24
	40	1.90	74075.6	1232.99
	60	2.80	100581.8	3534.05
	80	3.20	78284.2	5338.70
	100	4.20	125064.8	5633.29
7	20	6.00	17603.0	15.57
	40	11.70	25606.0	39.72
	60	16.20	27932.6	49.19
	80	23.70	43518.4	85.16
	100	27.70	43569.0	103.86
8	20	6.10	17617.4	19.52
	40	11.80	25901.8	48.45
	60	16.80	31955.9	84.61
	80	22.80	37124.2	120.26
	100	28.60	49807.5	178.07
9	20	14.40	54918.7	15.25
	40	28.00	104341.0	39.77
	60	43.90	167188.6	83.75
	80	57.80	212771.2	120.46
	100	69.60	250725.7	180.79
10	20	4.40	11274.7	16.01
	40	7.70	13784.3	26.26
	60	10.80	18085.2	66.19
	80	13.50	17828.5	83.46
	100	16.40	16952.9	138.27

Table 3: First-Fit Decreasing

5.2. Local Search

For our study we tested the local search algorithm with three different initial solutions, a) order the given piece's list in a non-increasing order (LSearchD); b) order the given piece's list in a non-decreasing order (LSearchA); and c) randomize the given list (LSearchR).

For each instance class we compare the average number of bins used and the average wasted space. The result are shown in Table 5 for the Berkey and Wang benchmark set and Table 6 for the Martelo and Vigo benchmark set. The Avg. Waste field shows the average empty area in the bins and the Time field shows the average time per instance in milliseconds.

As we can see in Table 5 and Table 6 the initial solution really influences the results of the heuristic. An initial solution in non-increasing order generates the result at least as good as the FFD heuristic.

The average times of this heuristic are higher than the FFD one and as we can see in Table 4, they go from 5 milliseconds to a little over 12 seconds, still none of the instances broke the soft limit of 300 seconds.

Class	N. Pieces	Avg. Bins	Avg. Waste	Avg. Time
	20	7.17	138.3	13.53
	40	13.60	199.9	34.84
1	60	20.10	217.1	84.56
	80	27.52	271.3	182.98
	100	32.30	232.4	286.82
	20	1.01	330.3	17.76
	40	2.00	639.9	92.75
2	60	2.80	727.1	226.71
	80	3.36	543.3	556.46
	100	4.09	683.4	880.79
	20	5.34	2452.2	50.15
	40	9.81	3363.1	239.40
3	60	14.31	3654.1	577.80
	80	19.63	4739.9	1163.70
	100	23.16	4920.2	1896.21
	20	1.00	3908.2	312.92
	40	2.01	7767.1	1955.33
4	60	2.81	8858.1	5449.77
	80	3.43	7631.9	13790.49
	100	4.20	9864.2	20517.10
	20	6.59	18069.5	260.80
	40	12.23	25375.6	1107.12
5	60	18.48	33381.8	2710.27
	80	25.00	40284.2	6449.03
	100	29.05	37564.8	11140.10
	20	1.00	42169.5	8373.08
	40	1.92	75875.6	40916.78
6	60	2.57	79881.8	61802.91
	80	3.12	71084.2	62344.01
	100	3.85	93564.8	63146.10

Class	N. Pieces	Avg. Bins	Avg. Waste	Avg. Time
	20	5.78	15403.0	284.41
	40	11.49	23506.0	1032.01
7	60	16.22	28132.6	2783.98
	80	23.39	40418.4	5632.54
	100	27.52	41769.0	9476.39
	20	5.98	16417.4	253.78
	40	11.56	23501.8	1373.60
8	60	16.67	30655.9	4310.05
	80	22.72	36324.2	8726.26
	100	28.19	45707.5	15648.64
	20	14.30	53918.7	187.62
	40	27.84	102741.0	1099.74
9	60	43.80	166188.6	3300.90
	80	57.74	212171.2	6185.63
	100	69.51	249825.7	11381.53
	20	4.37	10974.7	191.34
	40	7.59	12684.3	807.33
10	60	10.55	15585.2	3525.55
	80	13.21	14928.5	6797.54
	100	16.34	16352.9	13533.16

Table 4: Local Search with Non-increasing Starting Order

Class	N.Pieces	LSearchD		LSearchA		LSearchR	
		Avg.Bins	Avg.Waste	Avg.Bins	Avg.Waste	Avg.Bins	Avg.Waste
	20	7.17	138.3	8.25	246.3	7.51	172.3
	40	13.60	199.9	15.93	432.9	14.32	271.9
1	60	20.10	217.1	24.55	662.1	21.42	349.1
	80	27.52	271.3	34.29	948.3	29.49	468.3
	100	32.30	232.4	40.97	1099.4	34.53	455.4
	20	1.01	330.3	1.05	366.3	1.03	348.3
	40	2.00	639.9	2.03	666.9	2.01	648.9
2	60	2.80	727.1	2.96	871.1	2.87	790.1
	80	3.36	543.3	3.94	1065.3	3.50	669.3
	100	4.09	683.4	4.45	1007.4	4.20	782.4
	20	5.34	2452.2	6.30	3988.2	5.63	2916.2
	40	9.81	3363.1	12.26	7283.1	10.56	4563.1
3	60	14.31	3654.1	19.09	11302.1	15.84	6102.1
	80	19.63	4739.9	26.51	15747.9	21.74	8115.9
	100	23.16	4920.2	31.95	18984.2	25.62	8856.2
	20	1.00	3908.2	1.00	3908.2	1.00	3908.2
	40	2.01	7767.1	2.09	8567.1	2.02	7867.1
4	60	2.81	8858.1	3.03	11058.1	2.88	9558.1
	80	3.43	7631.9	4.07	14031.9	3.60	9331.9
	100	4.20	9864.2	4.50	12864.2	4.20	9864.2
	20	6.59	18069.5	7.63	28469.5	6.86	20769.5
	40	12.23	25375.6	14.85	51575.6	13.04	33475.6
5	60	18.48	33381.8	23.21	80681.8	19.87	47281.8
	80	25.00	40284.2	31.97	109984.2	27.14	61684.2
	100	29.05	37564.8	37.52	122264.8	31.64	63464.8
	20	1.00	42169.5	1.00	42169.5	1.00	42169.5
	40	1.92	75875.6	2.00	83075.6	1.94	77675.6
6	60	2.57	79881.8	2.88	107781.8	2.54	77181.8
	80	3.12	71084.2	3.81	133184.2	3.29	86384.2
	100	3.85	93564.8	4.46	148464.8	3.96	103464.8

Table 5: Comparing three versions of Local Search. Class 1-6

Class	N.Pieces	LSearchD		LSearchA		LSearchR	
		Avg.Bins	Avg.Waste	Avg.Bins	Avg.Waste	Avg.Bins	Avg.Waste
7	20	5.78	15403.0	6.20	19603.0	5.86	16203.0
	40	11.49	23506.0	12.85	37106.0	11.98	28406.0
	60	16.22	28132.6	18.57	51632.6	16.93	35232.6
	80	23.39	40418.4	27.11	77618.4	24.28	49318.4
	100	27.52	41769.0	32.04	86969.0	28.75	54069.0
8	20	5.98	16417.4	6.31	19717.4	6.08	17417.4
	40	11.56	23501.8	13.09	38801.8	12.07	28601.8
	60	16.67	30655.9	18.77	51655.9	17.03	34255.9
	80	22.72	36324.2	26.35	72624.2	23.76	46724.2
	100	28.19	45707.5	32.35	87307.5	29.29	56707.5
9	20	14.30	53918.7	14.83	59218.7	14.35	54418.7
	40	27.84	102741.0	29.12	115541.0	27.93	103641.0
	60	43.80	166188.6	45.75	185688.6	43.83	166488.6
	80	57.74	212171.2	60.35	238271.2	57.89	213671.2
	100	69.51	249825.7	72.98	284525.7	69.70	251725.7
10	20	4.37	10974.7	4.94	16674.7	4.61	13374.7
	40	7.59	12684.3	9.47	31484.3	8.36	20384.3
	60	10.55	15585.2	13.35	43585.2	11.39	23985.2
	80	13.21	14928.5	17.11	53928.5	14.41	26928.5
	100	16.34	16352.9	21.28	65752.9	17.95	32452.9

Table 6: Comparing three versions of Local Search. Class 7-10

5.3. Simulated Annealing

As we can see in Table 1 and Table 2, the results for the Simulated Annealing were better than the FFD and the Local Search but these results come at the cost of time. As shown in Table 7, the average time of each SA is higher than either of the previous ones. The time goes from 1,3 seconds to 74 seconds, hitting the soft limit of 150 seconds *per run* in 31 out of 50 instance type.

Since Simulated Annealing is heavily based on a stochastic method to get a new neighbor we have to run more than once and get the best result. Usually between 10 and 20 random seeds should be used to get better results. In our case we used 10 random seeds.

The tables 8 and 9 show the comparison in results from 3 different values of α: 90%; 95% and 99%. As explained in section 3 the α is the freezing rate such as a rate of 90% freezes faster than a rate of 99% where the temperature only decreases 1%. The freezing rate also influences the time the implementation runs, since it freezes faster, an implementation with α=90% will end faster than an implementation with α=95%. In our results α=95% is always better or equal to the results of α=90%. This is due to the fact that since we freeze slower we can accept worse solutions for longer time. Our results show that the implementation with α=99% sometimes gives worse results than the implementation with α=95. This happens because the soft time limit of 300 seconds is probably too small for an α this high. Given enough time, a better solution than the α=95% would have been found.

Class	N. Pieces	Avg. Bins	Avg. Waste	Avg. Time
1	20	7.10	131.3	1325.84
	40	13.41	180.9	6419.63
	60	20.08	215.1	17806.24
	80	27.50	269.3	36816.86
	100	32.18	220.4	64388.78
2	20	1.00	321.3	3309.99
	40	2.00	639.9	16163.95
	60	2.62	565.1	42686.36
	80	3.29	480.3	84034.56
	100	4.00	602.4	141117.44
3	20	5.13	2116.2	8734.18
	40	9.60	3027.1	42355.94
	60	14.10	3318.1	104469.92
	80	19.68	4819.9	150675.43
	100	23.41	5320.2	151417.94
4	20	1.00	3908.2	71984.58
	40	1.99	7567.1	151616.90
	60	2.78	8558.1	154080.09
	80	3.41	7431.9	157590.48
	100	4.20	9864.2	160839.20
5	20	6.50	17169.5	65160.42
	40	12.18	24875.6	151503.66
	60	18.59	34481.8	153955.53
	80	25.21	42384.2	157040.92
	100	29.27	39764.8	160730.49
6	20	1.00	42169.5	163208.78
	40	1.90	74075.6	195244.37
	60	2.50	73581.8	213891.78
	80	3.14	72884.2	401944.05
	100	4.13	118764.8	617682.24
7	20	5.51	12703.0	45783.92
	40	11.50	23606.0	150241.80
	60	16.14	27332.6	153257.31
	80	23.44	40918.4	156683.91
	100	27.51	41669.0	158556.04
8	20	5.80	14617.4	50658.42
	40	11.47	22601.8	151186.30
	60	16.60	29955.9	154135.21
	80	22.80	37124.2	156317.89
	100	28.24	46207.5	160663.47
9	20	14.30	53918.7	57851.73
	40	27.80	102341.0	151347.30
	60	43.70	165188.6	154939.95
	80	57.70	211771.2	158802.16
	100	69.50	249725.7	166050.36
10	20	4.35	10774.7	70110.31
	40	7.64	13184.3	151571.49
	60	10.55	15585.2	156028.11
	80	13.39	16728.5	158762.50
	100	16.39	16852.9	164043.74

Table 7: Simulated Annealing for a = 90%

Class	N.Pieces	SA90		SA95		SA99	
		Avg.Bins	Avg.Waste	Avg.Bins	Avg.Waste	Avg.Bins	Avg.Waste
1	20	7.10	131.3	7.10	131.3	7.10	131.3
	40	13.41	180.9	13.41	180.9	13.40	179.9
	60	20.08	215.1	20.09	216.1	20.04	211.1
	80	27.50	269.3	27.50	269.3	27.50	269.3
	100	32.18	220.4	32.04	206.4	32.31	233.4
2	20	1.00	321.3	1.00	321.3	1.00	321.3
	40	2.00	639.9	2.00	639.9	1.98	621.9
	60	2.62	565.1	2.66	601.1	2.70	637.1
	80	3.29	480.3	3.31	498.3	3.31	498.3
	100	4.00	602.4	4.14	728.4	4.18	764.4
3	20	5.13	2116.2	5.10	2068.2	5.10	2068.2
	40	9.60	3027.1	9.61	3043.1	9.72	3219.1
	60	14.10	3318.1	14.38	3766.1	14.48	3926.1
	80	19.68	4819.9	19.68	4819.9	19.69	4835.9
	100	23.41	5320.2	23.41	5320.2	23.41	5320.2
4	20	1.00	3908.2	1.00	3908.2	1.00	3908.2
	40	1.99	7567.1	1.97	7367.1	1.99	7567.1
	60	2.78	8558.1	2.79	8658.1	2.78	8558.1
	80	3.41	7431.9	3.41	7431.9	3.41	7431.9
	100	4.20	9864.2	4.20	9864.2	4.20	9864.2
5	20	6.50	17169.5	6.51	17269.5	6.52	17369.5
	40	12.18	24875.6	12.17	24775.6	12.19	24975.6
	60	18.59	34481.8	18.59	34481.8	18.59	34481.8
	80	25.21	42384.2	25.20	42284.2	25.21	42384.2
	100	29.27	39764.8	29.22	39264.8	29.27	39764.8
6	20	1.00	42169.5	1.00	42169.5	1.00	42169.5
	40	1.90	74075.6	1.90	74075.6	1.90	74075.6
	60	2.50	73581.8	2.67	88881.8	2.66	87981.8
	80	3.14	72884.2	3.20	78284.2	3.20	78284.2
	100	4.13	118764.8	4.20	125064.8	4.20	125064.8

Table 8: Comparing Simulated Annealing with different a. Class 1-6

5.4. Genetic Algorithm

As shown in Table 1 and Table 2, Genetic algorithm has generally better results than the FFD and LSearchD. But when compared to SA the results are a bit worse. The Class 9 problems give exactly the same results and Class 6, 7 and 8 instances with 100 pieces have on average a better result with GA, the difference is very small but GA takes almost 10 seconds more, on average, than the SA algorithm.

Looking at Table 10, we can see that the average time is higher than the remaining heuristics. We did not impose a time limit as with the SA implementation.

The tables 11 and 12 show the comparison in results from 3 different configurations: a) a mutation and survivability rates of 10%; b) a mutation rate of 75% and survivability rate of 10%; c) and a mutation rate of 10% and survivability rate of 75%.

We used a fixed number of generations and population size for all configurations, 25 generations and 50 chromosomes per population. The survivability rate is the probability that a chromosome will be destroyed to create new chromosomes through crossover or will pass unchanged into the next generation. The mutation rate is the probability of a mutation occurring in a chromosome. The algorithm we implemented for a mutation was the same we implemented on the neighborhood function in simulated annealing and local search, we swap the position of two pieces. All chromosomes can suffer a mutation. The average generation field is the generation in which the best solution was found.

The results for the GA are very similar in all configurations although the configuration with a higher mutation rate has, most of the time, a better result than the one with a lower mutation rate.

Class	N. Pieces	Avg. Bins	Avg. Waste	Avg. Time
1	20	7.10	131.3	712.48
	40	13.65	204.9	1452.53
	60	20.56	263.1	2381.62
	80	28.35	354.3	3382.10
	100	33.32	334.4	4363.03
2	20	1.00	321.3	3131.09
	40	2.00	639.9	6185.48
	60	2.78	709.1	9630.96
	80	3.40	579.3	13569.21
	100	4.20	782.4	16645.97
3	20	5.27	2340.2	9622.58
	40	9.82	3379.1	19064.89
	60	14.87	4550.1	27909.51
	80	20.66	6387.9	37971.64
	100	24.34	6808.2	49239.42
4	20	1.00	3908.2	77700.57
	40	1.99	7567.1	158595.00
	60	2.80	8758.1	249093.22
	80	3.40	7331.9	350542.72
	100	4.20	9864.2	439084.28
5	20	6.55	17669.5	80209.71
	40	12.27	25775.6	158303.37
	60	18.76	36181.8	231791.36
	80	25.82	48484.2	299942.59
	100	30.31	50164.8	399247.63
6	20	1.00	42169.5	346883.77
	40	1.90	74075.6	395092.35
	60	2.48	71781.8	454461.10
	80	3.16	74684.2	545691.98
	100	3.80	89064.8	541834.84
7	20	5.51	12703.0	30577.05
	40	11.51	23706.0	59131.90
	60	16.21	28032.6	110199.64
	80	23.63	42818.4	125799.63
	100	27.99	46469.0	179533.30
8	20	5.82	14817.4	31408.20
	40	11.52	23101.8	71406.25
	60	16.63	30255.9	118108.08
	80	22.94	38524.2	151200.84
	100	28.40	47807.5	198317.76
9	20	14.30	53918.7	22187.33
	40	27.80	102341.0	58332.75
	60	43.70	165188.6	115844.54
	80	57.70	211771.2	161607.92
	100	69.50	249725.7	234448.85
10	20	4.38	11074.7	57387.66
	40	7.80	14784.3	105950.88
	60	10.76	17685.2	179163.66
	80	13.83	21128.5	243539.11
	100	17.22	25152.9	284463.21

Table 9: Genetic Algorithm

Class	N.Pieces	Genetic			Survival			Mutation		
		Avg. Bins	Avg. Waste	Avg. Gen	Avg. Bins	Avg. Waste	Avg. Gen	Avg. Bins	Avg. Waste	Avg. Gen
	20	7.10	131.3	25.7	7.10	131.3	13.9	7.10	131.3	10.7
	40	13.65	204.9	24.5	13.70	209.9	18.7	13.71	210.9	15.4
1	60	20.56	263.1	26.0	20.71	278.1	24.4	20.51	258.1	22.7
	80	28.35	354.3	22.9	28.40	359.3	23.3	28.32	351.3	22.8
	100	33.32	334.4	24.6	33.39	341.4	23.6	33.37	339.4	23.2
	20	1.00	321.3	21.0	1.00	321.3	0.0	1.00	321.3	0.0
	40	2.00	639.9	24.0	2.00	639.9	24.4	2.00	639.9	22.0
2	60	2.78	709.1	25.3	2.77	700.1	24.8	2.81	736.1	23.6
	80	3.40	579.3	23.5	3.40	579.3	23.6	3.38	561.3	23.8
	100	4.20	782.4	27.4	4.20	782.4	24.2	4.20	782.4	23.9
	20	5.27	2340.2	24.9	5.30	2388.2	24.5	5.24	2292.2	18.7
	40	9.82	3379.1	24.1	9.87	3459.1	20.5	9.84	3411.1	23.0
3	60	14.87	4550.1	23.9	14.94	4662.1	24.6	14.86	4534.1	24.0
	80	20.66	6387.9	24.4	20.70	6451.9	26.3	20.57	6243.9	25.3
	100	24.34	6808.2	25.8	24.48	7032.2	24.5	24.35	6824.2	24.7
	20	1.00	3908.2	23.9	1.00	3908.2	21.2	1.00	3908.2	16.9
	40	1.99	7567.1	25.4	1.98	7467.1	23.9	2.00	7667.1	25.7
4	60	2.80	8758.1	24.7	2.79	8658.1	26.4	2.80	8758.1	24.5
	80	3.40	7331.9	24.2	3.41	7431.9	23.9	3.40	7331.9	21.8
	100	4.20	9864.2	25.2	4.20	9864.2	21.9	4.20	9864.2	16.2
	20	6.55	17669.5	25.9	6.57	17869.5	24.1	6.55	17669.5	19.3
	40	12.27	25775.6	23.4	12.30	26075.6	23.6	12.27	25775.6	24.2
5	60	18.76	36181.8	22.8	18.85	37081.8	24.5	18.78	36381.8	22.3
	80	25.82	48484.2	24.5	26.07	50984.2	25.7	25.91	49384.2	20.0
	100	30.31	50164.8	21.4	30.55	52564.8	24.3	30.48	51864.8	13.5
	20	1.00	42169.5	5.8	1.00	42169.5	10.5	1.00	42169.5	5.5
	40	1.90	74075.6	2.8	1.90	74075.6	4.7	1.90	74075.6	2.9
6	60	2.48	71781.8	1.8	2.49	72681.8	3.2	2.47	70881.8	1.5
	80	3.16	74684.2	1.3	3.12	71084.2	2.2	3.12	71084.2	1.2
	100	3.80	89064.8	0.8	3.76	85464.8	1.9	3.76	85464.8	1.0

Table 10: Comparing Genetic Algorithm. Class 1-6

Class	N.Pieces	Genetic			Survival			Mutation		
		Avg. Bins	Avg. Waste	Avg. Gen	Avg. Bins	Avg. Waste	Avg. Gen	Avg. Bins	Avg. Waste	Avg. Gen
	20	5.51	12703.0	26.6	5.56	13203.0	22.8	5.51	12703.0	25.5
	40	11.51	23706.0	24.5	11.52	23806.0	24.2	11.51	23706.0	27.0
7	60	16.21	28032.6	21.6	16.26	28532.6	24.4	16.21	28032.6	24.4
	80	23.63	42818.4	22.3	23.67	43218.4	22.2	23.59	42418.4	26.1
	100	27.99	46469.0	24.2	28.04	46969.0	24.1	27.99	46469.0	22.2
	20	5.82	14817.4	21.3	5.84	15017.4	20.8	5.82	14817.4	22.8
	40	11.52	23101.8	23.5	11.54	23301.8	22.5	11.50	22901.8	26.2
8	60	16.63	30255.9	22.2	16.63	30255.9	24.3	16.56	29555.9	23.9
	80	22.94	38524.2	26.8	22.98	38924.2	23.9	22.92	38324.2	21.5
	100	28.40	47807.5	25.2	28.43	48107.5	25.0	28.44	48207.5	14.6
	20	14.30	53918.7	21.9	14.30	53918.7	23.8	14.30	53918.7	18.0
	40	27.80	102341.0	25.3	27.80	102341.0	24.5	27.80	102341.0	25.2
9	60	43.70	165188.6	25.2	43.70	165188.6	24.0	43.70	165188.6	20.1
	80	57.70	211771.2	25.7	57.70	211771.2	23.2	57.70	211771.2	15.4
	100	69.50	249725.7	22.7	69.50	249725.7	24.2	69.50	249725.7	10.5
	20	4.38	11074.7	25.1	4.40	11274.7	25.1	4.36	10874.7	23.5
	40	7.80	14784.3	23.2	7.82	14984.3	26.1	7.79	14684.3	26.1
10	60	10.76	17685.2	25.3	10.86	18685.2	22.4	10.73	17385.2	21.9
	80	13.83	21128.5	23.6	13.95	22328.5	21.9	13.87	21528.5	18.2
	100	17.22	25152.9	22.6	17.25	25452.9	24.6	17.22	25152.9	15.4

Table 11: Comparing Genetic Algorithm. Class 7-10

6. Conclusion

We implemented one heuristic and three meta-heuristic, a constructive, a local search, a simulated annealing and genetic algorithm to solve the Rectangular Bin Packing Problem. We analyzed them using the most common Benchmark Sets in the literature for the problem defined on section 2: Berkey and Wang; and from Martello and Vigo.

One conclusion we can take from the results is that the way we pick the neighborhood in the local search heuristic is very prone to local optimal so starting with a good solution for our packing algorithm improves the chances of our heuristic finding a good solution.

In future tests, we could change the formula for the neighborhood and make it less prone to local optima, either by increasing the neighborhood size or by picking the best neighbor or a random best instead of picking the first best. In Simulated Annealing we could also increase the soft limit to 300 seconds and compare the different α's again, doing so would probably find a better solution with α=99%. In the case of the Genetic algorithm we could increase the population size and the generation number to give more time for the population to evolve and check how it affects the results.

As a final note, the comparison between these heuristics also shows that finding a solution can be very fast using a constructive heuristic like the First-Fit Decreasing. Improving on the other hand, can be very resource consuming as in the case of Local Search, Simulated Annealing and Genetic algorithm, where we already start with one or more feasible solutions.

References

Alvarez-Valdés, Ramón, Francisco Parreño, and José Manuel Tamarit. 2013. "A GRASP/Path Relinking Algorithm for Two-and Three-Dimensional Multiple Bin-Size Bin Packing Problems." *Computers & Operations Research* 40 (12). Elsevier: 3081-90.

Bays, Carter. 1977. "A Comparison of next-Fit, First-Fit, and Best-Fit." *Communications of the ACM*. DOI: 10.1145/359436.359453.

Berkey, J O, and P Y Wang. 1987. "Two-Dimensional Finite Bin-Packing Algorithms." *Journal of the Operational Research Society*. JSTOR, 423-29.

Blum, Christian, and Verena Schmid. 2013. "Solving the 2D Bin Packing Problem by Means of a Hybrid Evolutionary Algorithm." *Procedia Computer Science* 18. Elsevier: 899-908.

Chu, P.C., and J.E. Beasley. 1998. "A Genetic Algorithm for the Multidimensional Knapsack Problem." *Journal of Heuristics* 4: 63-86. DOI: 10.1023/A:1009642405419.

Dyckhoff, Harald. 1990. "A Typology of Cutting and Packing Problems." *European Journal of Operational Research* 44 (2): 145-59. DOI: 10.1016/0377-2217(90)90350-K.

Hong, Shaohui, Defu Zhang, Hoong Chuin Lau, XiangXiang Zeng, and Yain-Whar Si. 2014. "A Hybrid Heuristic Algorithm for the 2D Variable-Sized Bin Packing Problem." *European Journal of Operational Research* 238 (1). Elsevier: 95-103.

Hopper, E., and B. C H Turton. 2001. "Empirical Investigation of Meta-Heuristic and Heuristic Algorithms for a 2D Packing Problem." *European Journal of Operational Research* 128: 34-57. DOI: 10.1016/S0377-2217(99)00357-4.

Kirkpatrick, S, C D Gelatt, and M P Vecchi. 1983. "Optimization by Simulated Annealing." *Science (New York, N.Y.)* 220: 671-80. DOI: 10.1126/science.220.4598.671.

Lodi, Andrea, Silvano Martello, and Michele Monaci. 2002. "Two-Dimensional Packing Problems: A Survey." *European Journal of Operational Research* 141 (2). Elsevier: 241-52.

Lodi, Andrea, Silvano Martello, and Daniele Vigo. 1999. “Heuristic and Metaheuristic Approaches for a Class of Two-Dimensional Bin Packing Problems.” *INFORMS Journal on Computing* 11 (4). INFORMS: 345-57.

———. 2002. “Recent Advances on Two-Dimensional Bin Packing Problems.” *Discrete Applied Mathematics* 123 (1). Elsevier: 379-96.

Martello, Silvano, and Daniele Vigo. 1998. “Exact Solution of the Two-Dimensional Finite Bin Packing Problem.” *Management Science* 44 (3). INFORMS: 388-99.

Michael, R Garey, and S Johnson David. 1979. “Computers and Intractability: A Guide to the Theory of NP-Completeness.” *WH Freeman & Co., San Francisco*.

Osogami, T., and H. Okano. 2003. “Local Search Algorithms for the Bin Packing Problem and Their Relationships to Various Construction Heuristics.” *Journal of Heuristics* 9: 29-49. DOI: 10.1023/A:1021837611236.

Pisinger, David, and Mikkel Sigurd. 2005. “The Two-Dimensional Bin Packing Problem with Variable Bin Sizes and Costs.” *Discrete Optimization* 2 (2). Elsevier: 154-67.

———. 2007. “Using Decomposition Techniques and Constraint Programming for Solving the Two-Dimensional Bin-Packing Problem.” *INFORMS Journal on Computing* 19 (1). INFORMS: 36-51.

Sarabian, M, and L V Lee. 2010. “A Modified Partially Mapped Multicrossover Genetic Algorithm for Two-Dimensional Bin Packing Problem.” *Journal of Mathematics and Statistics* 6 (2): 157.

Wäscher, Gerhard, Heike Haußner, and Holger Schumann. 2007. “An Improved Typology of Cutting and Packing Problems.” *European Journal of Operational Research* 183 (3): 1109-30. DOI: 10.1016/j.ejor.2005.12.047.

Wei, Lijun, Wee-Chong Oon, Wenbin Zhu, and Andrew Lim. 2013. “A Goal-Driven Approach to the 2D Bin Packing and Variable-Sized Bin Packing Problems.” *European Journal of Operational Research* 224 (1). Elsevier: 110-21.