Efficient algorithms for real-life instances of the variable size bin packing problem

Bang-Jensen, Jørgen; Larsen, Rune

Published in:
Computers and Operations Research

Publication date:
2012

Document Version
Early version, also known as pre-print

Efficient Algorithms for Real-Life Instances of the Variable Size Bin Packing Problem

Jørgen Bang-Jensen[1], Rune Larsen[1]

[a]*Department of Mathematics and Computer Science, University of Southern Denmark*

Abstract

In this paper we present a local search heuristic for real-life instances of the variable size bin packing problem, and an exact algorithm for small instances. One important issue our heuristic is able to satisfy is that solutions must be delivered within (milli)seconds and that the solution methods should be robust to last minute changes in the data. Furthermore we show that we are able to incorporate the concept of usable leftovers on a single bin, and the implementation of many additional constraints should be supported by the straightforward solution representation. The heuristic is compared to others from the literature, and come out ahead on a large subset of the instances.

Keywords: Variable Size Bin Packing Problem, Cutting Stock, Real-Life Application, Heuristics

1. Introduction

Several real-life problems can be formulated as packing or cutting problems or relaxed versions thereof. Therefore packing and cutting problems has been studied extensively, and since Gilmore and Gomery published their integer programming based approach [1], most methods have been based on this idea. The method is based on solving an integer programming problem with a reduced set of columns each corresponding to some packing patterns of a single bin, and generating additional columns as necessary.

This method does not perform as well on bin packing problems where the bin size might vary, as increasing the number of bin sizes, increases the

Email addresses: jbj@imada.sdu.dk (Jørgen Bang-Jensen), rular@imada.sdu.dk (Rune Larsen)

number of columns that must be added to the basis set before an optimal solution of the lp relaxation is reached. An overview of the literature on the variable size bin packing problem (VSBPP) can be found in [2] and [3], and most employed methods are based on Gilmore and Gomerys work.

The common approaches suffer form multiple potential drawbacks when considering real-life instances coming from industrial companies:

- Some companies cannot guarantee a single bin size, but require the size to vary from bin to bin, thus increasing the number of packing patterns dramatically. For example, this situation can arise due to deformation of the material during or after a casting process.
- Some companies require that the code can be run on embedded systems with very limited memory and sometimes even online memory allocation, thus excluding the use of LP solvers.
- Some companies gain more knowledge about the problem data online, and require this information to be taken into account immediately. They might also require a solution within milliseconds while allowing a possible negative impact on its quality.
- Some companies are capable of reusing unused capacity in the last bin packed. This situation often arises when modeling a cutting problem, where cutting can be resumed on the last used item, given that it is large enough to warrant storage.

Motivated by these difficulties, our aim was to develop and implement a construction heuristic and a local search heuristic, which could produce high quality solutions very fast, when a large number of bin sizes are available and at the same time the algorithms should adapt fast to online changes in data.

1.1. Two Real Life Problems

Our work is motivated by the following real-life problems which we have encountered through collaboration with industrial companies.

1.1.1. An Anonymous Company

The first company produces machinery that uses a combination scale to create packages of certain weights based on a number of smaller compartments containing material of known weights. At any given point in time, they can ask for a certain weight of material to be released, and the machine

should open the combination of compartments creating the closest possible weight that exceeds the requested amount. After this the compartments just emptied are refilled and their new weight is determined.

A further complication arises when some compartments are not filled before the next release of material. When this can be predicted, we wish to ensure that the compartments that are released leaves a set of filled compartments with the best possible options for new compartments to release. This corresponds to calculating a packing of two bins, where the wasted space in the first is considered more important than that of the second. This problem is relatively easy except for the fact that in one of the real-life applications we consider it is extremely time-critical, and can never take in excess of 100 milliseconds even on an embedded processor.

The first problem is easily solvable, using an optimized version of the dynamic programming heuristic described in [4] which is usually employed to solve subset sum problems. While the second problem basically corresponds to a bin packing problem.

1.1.2. Danfoam

The company Danfoam A/S produces high quality foam mattresses for a world wide market. The first part of the production involves cutting prescribed length foam blocks (typically from 110 to 230 cm) from larger blocks. These larger blocks which reside in a storage are either previously unused blocks whose lengths are around 30 m, or they are remainders from a previous cutting process in which case the length lies between 8 m and 29 m.

The exact data (available to us and used in some of the tests) cannot be described here due to non-disclosure agreements, but the overall structure of the daily cutting problem is as follows: A substantial number (several hundreds) of items whose lengths are in the interval between 110 cm to 230 cm (about 10 different lengths) are to be cut from blocks residing in the storage. These block lengths will vary due to setting after the casting process, and possibly previous cuts taken of that block. The amount of waste should be minimized with the additional condition that the last block used may be returned to storage if it is at least a certain length, say 8 m (in which case no waste is counted for this block).

A further complication arising when optimizing the cutting pattern is that, due to the properties of the foam material, the length of a block might change slightly as it is brought in for cutting. Furthermore some items might be damaged during production and have to be rescheduled on the remaining

blocks to be cut.

Thus overall we are faced with a variable sized bin packing problem with a large number of different bin sizes which are typically much larger than the item sizes (usually 10-15 items or more fit in a random bin) and where there are relatively few item sizes (about 2-10 different in the range 110 cm to 230 cm). Due to the online changes in the data, the time requirements are strict.

2. Results From the Literature

The literature concerning cutting and packing is split due to the difference in characteristics of the problems considered, inferred by small variations in the input data. Classification schemes were proposed in [5] naming the problem 1/V/D/R or 1/V/D/M, but the scheme was found lacking and expanded upon in [3] naming the problem a Residual Bin Packing Problem or Residual Cutting Stock Problem. This demonstrates a major problem when classifying real life problem types and solvers efficiency: A real life problem might well contain instances that are of widely different types, and a solver might be usable on a wide array of different types of problems.

The following characteristics are significant when considering a problem:[1]

- Input minimization vs. output maximization.
- Objective function strongly correlated/equal to wasted residual objects vs. objective function weakly correlated or independent from amount of wasted residual objects.
- Homogeneous objects vs. heterogeneous objects.
- Homogeneous items vs. heterogeneous items.
- Low average number of items per object vs. a high number of items per object in a good solution.

Besides this, real life problems usually contains side constraints such as:

- No waste can be reused later and thus ignored vs. a set number of items with waste above a certain length can be ignored vs. all waste

[1] Objects are the bins or rolls, and items are the items or cuts in the packing and cutting terminology respectively.

above a certain length can be ignored. These concepts are often called usable leftover in the literature such as [6] [7] etc.

- Short time available for solving vs. long time available for solving.

When considering real life problems one should specify what range the above characteristics can occur in, and likewise when talking about a solution method, it should be stated which ranges of the above characteristics it can be brought to work on, and how it might influence performance.

The Danfoam problem is an input minimization problem with heterogeneous objects and instances with either heterogeneous or homogeneous items. It has a relatively high number of average items per object (10-15), one can ignore waste in *one* of the bins if the waste is large enough, and instances must be solved in a very short time frame.

The exact heuristics have grown increasingly promising for similar problems as seen in [8], [9] and most recently [10], but as cuts and bounds become more advanced, factoring in the possible waste on a single bin and similar real-life constraints, become more intractable.

Recent work on heuristics for the variable size bin packing problem [11] has shown some impressive results. Problems are solved extremely fast, and often to optimality. However the problems considered in this paper do have a few vital differences from the ones considered in [11] and other articles.

- Bins can contain a larger number of items, massively increasing the number of columns a column generation based solver would have to create. A genetic algorithm would have to increase population size significantly, to make sure that all possible assignments of items to bins can be created by crossovers.

- Bins are very unlikely to share size with other bins, creating problems similar to the mentioned above.

- Any algorithm is *required* to be able to deliver an answer anytime after a very short initialization time, thus preventing a lot of column generation approaches from producing solutions in time. This includes the demand, that the algorithm must be able to resume optimization fast, given any changes in either bin or item sizes.

- The number of item sizes are limited, which often requires a change of assignments for a large number of items, before a better solution can be reached.

- In one of our applications, waste concentrated on the last bin can be safely ignored, as long as it remains above a certain size.

Other recent work [12] [13] has been concerned with real life problems where leftover was considered usable, but they allow waste on all bins, thus enabling the integration of this side constraint into a column generation scheme. The same problem is encountered when considering the paper [7]. The paper [6] does consider the case where the number of usable leftovers is bounded, but it allows only a closed set of leftover lengths, and bounds on the number of each of these separately to integrate them into an integer program.

3. Notation and Problem Definition

The bin packing problem considered is as follows:

Given k bins $B_1 \dots B_k$ of sizes $v_1 \dots v_k$ respectively, and n items x_1, $x_2 \dots$ x_n of sizes $a_1 \dots a_n$ respectively: Find a partition of items $x_1 \dots x_n$ into sets $S_1, S_2, \dots, S_k \in S$ such that $\sum_{x_i \in S_j} a_i \leq v_j$ for $j = 1 \dots k$.

Two variants of the problem are considered in this paper, and the objective function depends on the variant under consideration.

3.1. Classical Bin Packing

In the classical bin packing problem BPP, all bins are of the same size v, thus $v_1 = v_2 = \ldots = v_k = v$. The wasted space, in any solution to this problem, depends only on the number of bins utilized. Thus the objective function normally just focuses on minimizing this number. Other papers such as [14] have employed more advanced objective functions, to distinguish between the attractiveness of solutions utilizing the same number of bins.

When using an algorithm that optimizes with respect to a different objective function f_a that does not match the actual objective function f, some properties can be helpful to maintain: Given two solutions A, B in a minimization problem.

- $f(A) < f(B) \iff f_a(A) < f_a(B)$ preserving a strict preference for better solutions.
- If $f(A) = f(B)$ and $f_a(A) < f_a(B)$ A should contain characteristics that are considered preferable or likely to lead to an improving solution.

- Evaluation of the contribution of each bin to f_a should be possible, thus allowing for Δ-evaluations.

In this paper, the quality of a solution is considered to be the sum of the waste in the solution, and ties are broken favoring the solution having the largest sum when summing up the waste per bin squared. This creates a situation where solutions with an equal amount of waste, are considered better the more the waste is concentrated. Experiments have verified this strategy to improve the local search.

3.2. A Generalization Allowing Varying Bin Sizes

In many real life instances such as the ones treated in this paper, the assumption that the bin size is constant does not hold. In that case $v_1, v_2, \ldots, v_k$ might differ, and the number of bins used fail to express the space wasted by an assignment of items to bins. In these cases the objective is often to reduce the sum of the unused capacity in the used bins.

Minimize $\sum_{S_j \in S} w_j$

where $w_j = \begin{cases} v_j - \sum_{x_i \in S_j} a_i & \text{if } |S_j| > 0 \\ 0 & \text{otherwise} \end{cases}$

For some real life instances, that arise from cutting stock like problems such as the one mentioned from Danfoam, the last bin packed can contain spare room, that is not considered waste, as it can be used in the next packing. This is not trivial to model when using the standard column generation based techniques, but as we will show later, it is achievable using the developed local search heuristic.

4. Subproblems

The bin packing problem can be seen as several interdependent subset sum problems. Several efficient methods, as described in [4], [15] and [16], exist for solving these given sufficiently small bins or items. As this is typically the case for bin packing problems, these methods can be used for exploration of neighborhoods and for construction heuristics, by providing optimal packings of a single bin.

4.1. A Single Bin

The simplest subproblem consists of assigning items to a single bin, without concern for subsequent assignments. This can be achieved by using

dynamic programming to compute all possible sums of item sizes. The sum closest to but no higher than the bin size is now chosen, and the items giving the sum is then found by backtracking in the dynamic programming table. This corresponds to construction heuristics used in [11], [17] and analyzed in depth in [18] and [19]. Our implementation differs from theirs, allowing for subsequent use in an exact solver.

Sum \ Item size	3	2	4	...	1
0	1	1	1	...	1
1	0	0	0	...	1
2	0	1	1	...	1
3	1	1	1	...	1
4	0	0	1	...	1
5	0	1	1	...	1
⋮	⋮	⋮	⋮	⋱	⋮
s	0	0	0	...	?

Table 1: Dynamic programming table. s is the maximum sum that is computed, which is usually the size of the bin to fill.

Table 1 demonstrates the concept. The first column signifies that the sums 0 and 3 can be made with the item seen so far. The second column that the sums $\{0, 2, 3, 5\}$ can be made with the two items seen so far.[2]

A sum can be achieved by combining a subset the items seen so far, if and only if the number in the corresponding row and the last column is 1. The sum can be achieved *without* the item at the head of the column if and only if the previous column has a one in the same row. Likewise the sum can be achieved *by using* the item corresponding to the column if and only if the preceding column has a 1 in the row x places above, where x is the size of the item corresponding to the column. Using this method recursively as shown in Algorithm 1 on page 10, all combinations of items yielding a given sum can be enumerated. Its worth noting that an exponential number of these might exist.

[2]The size of the item is shown above each column.

4.1.1. Favoring Use of Large Items

Larger items often makes fitting items into bins harder.[3] To favor combinations using these large items the following strategy has been employed:

Before construction of the dynamic programming table, the items are sorted according to size in increasing order. During backtracking, items are thus encountered in decreasing order, and the algorithm simply chooses an item, if it can be part of the desired sum. Similar preferential treatment for larger items has been suggested in [11], [17] and [20] amongst others.

4.1.2. Avoiding Duplicate Subsets

A common problem encountered when working with sets of numbers, is that the number of representations of the same set can explode when multiple items have the same size.

Lets say we have 5 items with the size a. If a set of items fills out a bin, and contains exactly one item of size a, we know that 4 more such sets exists that uses the other items of size a. From a bin packing point of view, these sets are no different from the one we have already seen, and we would like to generate only one of these sets. This is done by sorting items before the dynamic programming table is created, and enforcing that once an item of a certain size is skipped during backtracking, all subsequent items with the same size must also be skipped. This ensures that if k items of size a is chosen, it will always be the first k in the ordering thus eliminating generation of equivalent solutions.

4.1.3. Optimizations

It should be noted that the dynamic programming table conceptually consists of boolean values, and that the operations computing a new column are the same for all its boolean entries. Namely $b_{i,j} = b_{i-1,j} \vee b_{i-1,j-x}$ where i indexes columns, j indexes rows, and x is the size of the item corresponding to the i'th column.[4]

The columns in the table can thus be represented by sets of integers corresponding to the value of each column interpreted as a bitstring, and all operations can be performed as bitwise boolean operations between the integers. This yields a significantly improved though asymtotical equivalent

[3] This is the motivation for applying the First fit *decreasing* heuristic.

[4] Out of bound values are assumed to be false.

Data: A dynamic programming table $dpt_{sum,i}$.
Item sizes $a_1 \dots a_n$ corresponding to each column as described in section 4.1 sorted in increasing order as described in section 4.1.1.
A maximum index of usable item m.
The last used item size l.
A desired sum s.
Result: A list of items that sums to s, or false if this is impossible.

```
if s = 0 then // We are done as we are trying to sum to 0
    return ∅;
end
if a_m ≥ s then // The item is too large to use
    return readSolution(dpt_{sum,i}, a_1 ... a_n, m − 1, l, s);
end
if m = 0 ∨ dpt_{s,m} = 0 then // No solution with the desired sum
    return False;
end

/* Can create a solution using the m'th item?                    */
if dpt_{s−a_m,m−1} = 1 ∧ l ≠ a_m then
    result ← readSolution(dpt_{sum,i}, a_1 ... a_n, m − 1, a_m, s − a_m);
    if result ≠ False then
        return a_m ∪ result;
    end
end
/* Can create a solution not using the m'th item?                */
if dpt_{s,m−1} = 1 then
    result ← readSolution(dpt_{sum,i}, a_1 ... a_n, m − 1, l, s);
    if result ≠ False then
        return a_m ∪ result;
    end
end
/* Only reachable if similar sized items were skipped.           */
return False;
```

Algorithm 1: readSolution($dpt_{sum,i}$,$a_1 \dots a_n$,m,l,s) The next solution can be obtained by blocking the use of the item a_i with the lowest i in the previous solution, and unblocking all a_j for $j < i$

running time in a critical region of the code. In our practical applications this optimization is of great importance.

4.2. An Exact Solver for VSBPP

As the bin packing problem is NP-hard, no polynomial time algorithm can be expected to solve the problem to optimality. For sufficiently small instances however, there might exist algorithms that finish quickly enough to be viable. The definition of 'quickly enough' varies from application to application. In some of the interactive applications, two seconds might be very acceptable, while a quarter of a second is pushing the limit when interacting with fast paced machinery. Recent work in the field of exact solvers includes the following articles mentioned in the literature review: [10] uses branch and bound to solve instances with an unlimited supply of each bin size and both convex and concave cost functions. [8] and [9] solves a multiple length cutting stock problem that is equivalent to the problem treated in this paper aside from the low number of cutting stock lengths, high number of item sizes, their relation between item and bin sizes and their inability to handle usable leftovers. If problems that have these characteristics are considered, the local search proposed can easily be adapted to use these as solvers instead.

Our exact solver (Algorithm 2 and 3) uses the dynamic programming approach to solve the subset sum subproblems, it then iterates through the solutions attempting to solve the induced problem recursively. To make the approach faster, several optimizations have been made:

- The iteration is performed in an order that ensures that once a feasible solution is found, it is optimal and the search can be halted.
- Several fail fast checks have been implemented, detecting that no solutions exists before all possible assignments have been enumerated.
- If there exists an upper bound b on the waste that the optimal solution can contain. The number of bins of certain size can be removed as long as enough bins of that size remain to contain all items plus b units of waste.

```
Data: A set Bins of k bin sizes v_1 ... v_k
A set Items of n item sizes a_1 ... a_n
A minimal amount of waste that can be ignored in a bin (minIgnored)
Result: An optimal partition of items into bins

Compute all possible sums of bin sizes v_1 ... v_k;
Compute all possible sums of item sizes a_1 ... a_n;
itemSum ← sum_{i=1}^{n} a_i;
maxSpill ← max(0, max(v_1 ... v_k) − minIgnored);
for waste ← 0 to infinity do
    for spill ← 0 to maxSpill do
        if (waste + itemSum − spill) is an obtainable sum of bin sizes
        and (spill) is an obtainable sum of item sizes then
            for Each binSet with sum of sizes
            (waste + itemSum − spill) do
                if max{v_i | v_i ∈ Bins \ binSet} − spill > minIgnored
                then
                    continue;
                end
                for Each itemSet with sum of sizes (itemSum − spill)
                do
                    solution ← attemptAssign(binSet,itemSet,waste);
                    if solution ≠ null then
                        assign (Items \ itemSum) to an unused bin in
                        solution;
                        return solution;
                    end
                end
            end
        end
    end
end
```

Algorithm 2: The algorithm searches for solutions with an increasing amount of waste. For each waste amount it searches through solutions assigning increasing amounts of items (*spill*) to the bin that has its waste ignored.

Data: A set $Bins$ of k bin sizes $v_1 \dots v_k$
A set $Items$ of n item sizes $a_1 \dots a_n$
An allowed amount of waste in the bins: $slack$
Result: An optimal partition of items into bins

if $k = 0$ **then** `// If we have filled all bins`
  return *Empty solution*;
end
$itemSums \leftarrow$ Compute all possible sums of item sizes $a_1 \dots a_n$;
if *failFastChecks(*$Bins, slack, itemSums$*)* **then**
  return *null*;
end
$surplus \leftarrow 0$;
while $surplus < v_1$ *and* $surplus < slack$ **do**
  for *Each set itemSet of items summing to* $v_1 - surplus$ **do**
    $solution \leftarrow$
    attemptAssign($bins \setminus v_1$,$Items \setminus itemSet$,$slack - surplus$);
    if $solution \neq null$ **then**
      return $solution \cup itemSetassignedtov_1$;
    end
  end
  $surplus \leftarrow surplus + 1$;
end
return *null*;

Algorithm 3: attemptAssign($Bins$,$Items$,$slack$) This algorithm assumes that all bins need to be used. The function failFastChecks is described in Section 4.2.2

4.2.1. Iteration Order

To ensure that the first feasible solution found is also optimal, a variable keeps track of the amount of wasted space allowed. This variable is initialized to 0. If the current bin cannot be filled causing less than that amount of wasted space, there is no solution with that amount of waste, and the subproblem is infeasible. If the initial problem is deemed infeasible, the waste variable is incremented, and the search repeated.

In the traditional case of a uniform bin size, a further optimization can be realized by ensuring that the waste plus the sum of the items always correspond to a multiple of the bin size.

4.2.2. Fail Fast Checks

The first fail fast optimization consists of checking whether the sum of item sizes plus the allowed waste, equals something obtainable by combining bins. For traditional bin packing problem this corresponds to checking if it equals a multiplum of the bin size, whereas it can be achieved by using the dynamic programming approach on the bin sizes if they are allowed to vary.

The second fail fast check is done each time items are assigned to a bin. If the minimum number of bins needed for the remaining items cannot be filled without incurring more than the alloted wasted space, the partial assignment can lead to no feasible solution. This check is made by relaxing the remaining problem to allow the use of all items in all bins, and can be done trivially using the dynamic programming table.

5. Solution methods for the BPP

As the goal was to create a method that did not rely on external solvers, and gave solutions superior to those obtainable with First Fit Decreasing (FFD) etc., local search was considered. Firstly an initial solution had to be constructed, and in addition to First Fit Decreasing and Best Fit, a dynamic programming approach as seen in [18] and [19] was considered as construction heuristic.

5.1. A Dynamic Programming Construction Heuristic

To generate initial solutions for the local search heuristic, a dynamic programming approach is used. First it generates all possible sums of item sizes, and greedily assigns the best fitting set of items to the first bin. Then it solves the remaining problem recursively.

For the problem faced by the undisclosed company, this construction heuristic produces guaranteed optimal combinations of compartments to release when considering a release event locally. When using the dynamic programming table, it's trivial to iterate through all combinations of compartments satisfying this. The ability to use the remainder of the compartments efficiently given a set of compartments to release, can be measured by rerunning this construction heuristic on the remaining compartments. Should this technique approach the time limit, the best found solution so far can always be returned instantly.

5.2. A Local Search Heuristic

The local search heuristic (DPLS) (as outlined in Algorithm 4) takes an initial solution generated by the dynamic programming construction heuristic. It then creates subproblems of a more manageable size, and tries to reoptimize them to optimality.

Other initial solution generation schemes can be chosen, but the algorithm performs better given a solution where as many bins as possible are packed very well, as opposed to having every bin packed reasonably well. Experiments showed that using FFD or similar construction heuristics for generating an initial solution decreased solution quality and/or increasing computation time on average.

Given a feasible solution to the problem, the heuristic first checks if there is wasted enough space to potentially use one bin less. If this is the case, the heuristic identifies the bins containing wasted space and chooses a subset of these again ensuring that they contain enough wasted space to potentially free one bin. Besides these sets, one or more full bins might also be chosen to increase the size of the neighborhood.

Given the subset X of bins found as above, the heuristic undoes all assignments of items to members of X, and attempts to reassign the items to the available bins[5] optimally using the exact solver. A new solution is accepted if it uses fewer bins, or if it concentrates the waste more.[6] The concentrated waste criteria was added to bring the local search out of local minimas, without requiring the release of a very large number of bins.

[5]That is, X and all previously unused bins

[6]If the sum of the squared waste in each bin is higher.

```
Data: An initial solution sol.
nbReleased ← 2;
while The allotted time has not expired and the solution is not
verifiably optimal do
    X ← selectNextSubset(nbReleased);
    subS ← reassign items from X optimally;
    if cost(subS) < cost(X) then
        Reassign items from X in sol according to their assignment in
        subS;
    end
    if All releasable subsets have been explored since last improvement
    then
        nbReleased ← nbReleased + 1;
    end
end
return sol
```

Algorithm 4: A general local search based on dynamic programming. The differences between the local search algorithms used in this paper consists in 1) The ways the subsets X are selected. 2) The check for whether optimality is achieved and 3) In the objective function used.

6. Generalizing the Method to Our Real-Life Application

In the real life problem of Danfoam, the bin sizes will vary and the optimization objective is to minimize the wasted bin space in the used bins, possibly ignoring waste in the last bin if it exceeds a certain amount. This requires adaptations in both the construction heuristics and the local search heuristic.

6.1. Dynamic Programming Construction Heuristic for VSBPP

The approach can easily be generalized to the case where bins vary in sizes: Instead of assigning to the first bin, the potentially wasted space is computed for each bin, and the one with the least amount of potentially wasted space is chosen. To break ties, the smaller or larger of the bins can be chosen, depending on what seems preferable in the problem.[7]

6.2. Dynamic Programming Based Local Search Heuristic for VSBPP

Like in the case with constant bin sizes, a set of bins are selected and all their assignments are released. The first motivator for change consists in the fact, that a better solution might be reached, even if the sum of the wasted bin space does not exceed the size of a bin. This can be seen by considering a case where a bin has wasted space, and a smaller bin exists that can contain the same items.

The second difference is that bins that are in use, and whose wasted space exceeds the size of the smallest released element can have its wasted space considered as a bin in its own right in the subproblem. This further reduces the need to release assignments on bins with wasted space.

To exploit these properties, relatively few bins are released initially, and the number is increased until the time allotted is spent. To demonstrate the consequences of various time limits, graphs of cost vs. time are presented in Figures 1 and 2 on page 24.

When adapting the objective function to the alternative one, where a bin with more waste than 800 cm can be considered having no waste, it is important to note that our exact solver will iterate through candidate solutions in a way that seeks to concentrate waste on the last bin considered. Thus when a subproblem is integrated into the current solution, it tends to

[7]Smaller bins are usually chosen, as they usually contain fewer items, and the possible sizes obtainable with fewer items tend to be fewer.

have the waste concentrated, and the local search algorithm (DPLSW) can just use the new objective function as acceptance criteria for any candidate solution.

7. Performance and Comparisons

Measuring the performance of the developed heuristic is not trivial, as each set of benchmark instances are tailored towards a specific configuration of the problem. The instances used in [14] can be found at the OR library [21], and they provide a reasonable estimate of performance on instances with the same bin to item size ratio even though they are traditional bin packing problems. The instances used in [11] differ significantly from the ones encountered in the real-life problems under consideration, in that it focuses on problems with very few items per bin, and a larger number of item sizes while still making waste in a large number of bins necessary. The instances from [22] have multiple bin sizes, and relatively few item sizes, but their relative size only matches the encountered real life problem on a few of the classes of problems defined.

Generators and instances used in this article, can be obtained by contacting the authors.[8]

Coding was done in Java 1.6 and in all tests, a single thread was used on a Core i7 920 processor having 4 gigabytes of ram, and unless otherwise noted, an upper bound of ten seconds time usage has been enforced on each instance. Given enough time, the algorithm can usually improve the solution, and given enough time and memory it will give the optimal solution, but for all but the smallest instances from [22] this has not been done. On the real life instances, graphs are presented to show the quality of the solutions as a function of the time used.

7.1. Dynamic Programming Construction Heuristic

The obvious benchmark for a construction heuristic, must be a test against the current most popular approach, which until lately was the first fit decreasing algorithm as seen in [20] and [22] amongst others.[9] The average[10]

[8]Note that the data from Danfoam is not freely available.

[9]In [11] and [17] they also propose subset sum, dynamic programming based solutions, and the method is rapidly gaining acceptance.

[10]The number of bins used, averaged over all instances.

number of bins used can be seen in Table 2, and can be compared to the best solution known. This shows that the dynamic programming algorithm outperforms FFD on average and gets very close to optimality. On the instances from [14], DPC and FFD finds the best solution in 133 and 2 of the 140 instances respectively, and tie on the remaining 5, thus we must conclude that DPC nearly always yields better solutions than FFD. This price of this improvement is that DPC does not run in polynomial time in the size of the bins, but this seems less alarming as the time it takes for both algorithms to finish all instances combined is measured in a few seconds. Furthermore it should be noted that the instances have a higher bin size than typically encountered as described in [23], and that the real life applications of DPC has proven its efficiency on much larger instances as seen in subsection 7.3.

The DPC matches the currently best known solution on 28 of the instances, and often this can be proven to be an optimal solution as relaxing the constraint that items cannot be partially assigned, gives the same cost in number of bins used.

Current best	DPC	FFD	DPLS
234.4833	235.15	237.4667	234.6667

Table 2: DPC VS. FFD average waste over all 140 instances. DPLS is described in subsection 7.2.

A generation scheme for test instances for general one-dimensional cutting stock problems can be found in [22], and Table 3 clearly shows that DPC outperforms the other presented constructive heuristics.

7.2. Dynamic Programming Based Local Search

As the performance of solvers is highly dependent on the characteristics of the problem, finding a fair basis for comparisons can be challenging. The public test instances having the most similar bin and item size ratio with the encountered real life problem from Danfoam, are the instances in [21]. They are however single size bin packing problems, and all results obtained should be treated accordingly. The proposed local search heuristic does however manage to improve on the results produced by the dynamic programming construction heuristic on the instances from [14], creating results that are only about 0.18 bins or 0,078% above the currently best known solution as seen in table 2.

Class	K	m	v_1	v_2	best constr.	DPC	best resi.
1	3	5	0.01	0.2	118.75	250.25	108.70
2	3	5	0.01	0.8	42429.35	1739.48	683.05
3	3	5	0.1	0.8	31493.55	2309.60	617.20
4	3	20	0.01	0.2	170.50	274.90	155.65
5	3	20	0.01	0.8	49565.05	5349.08	392.60
6	3	20	0.1	0.8	84775.60	6016.45	361.40
7	5	10	0.01	0.2	152.30	303.38	129.20
8	5	10	0.01	0.8	73920.20	3821.23	545.00
9	5	10	0.1	0.8	96475.80	3780.48	785.10
10	5	20	0.01	0.2	163.80	293.65	147.60
11	5	20	0.01	0.8	81946.20	5943.08	172.05
12	5	20	0.1	0.8	98766.15	7733.68	192.90
13	7	10	0.01	0.2	146.05	380.38	115.90
14	7	10	0.01	0.8	87379.30	3342.33	241.35
15	7	10	0.1	0.8	118647.90	4018.05	247.65
16	7	20	0.01	0.2	190.30	511.78	126.00
17	7	20	0.01	0.8	91638.20	6751.58	132.60
18	7	20	0.1	0.8	125865.15	6754.70	153.65

Table 3: Testruns on instances from [22], K is the number of bin sizes, m is the number of item lengths, and v_1 and v_2 are the upper and lower bounds on item sizes relative to average bin size. Bin sizes are chosen uniformly randomly between 100 and 1000. The best results of the constructive and the residual algorithms presented in [22] can be seen in the columns best constr. and best resi. respectively.

Table 3 and 4 contains results based on 400 instances per class generated as described in [22][11] They have multiple though fairly few "bin" sizes, but they suffer from a smaller item to bin size ratio on many of the classes. On the instances with the highest bin to item size ratio, the proposed local search consistently outperforms the heuristics proposed in [22] within a quater of a second, and on many of the remaining instances the local search delivers better or comparable results fast, as can be seen in Table 4. The running time in [22] was an *average* of 2.27 to 2.59 seconds on a Pentium III (866 MHz

[11]This scheme might generate infeasible instances. These have been proven infeasible and then discarded.

- 256 MB RAM) depending on the heuristic. As the Core i7 920 processor and its 4 GB RAM is significantly more powerful, a *maximum* running time of 0.25 seconds in Table 4 is the fairest column to compare to, but the ability of DPLS to make use of extra time should be taken into consideration.

The actual 40 instances per class of [22] were unobtainable, and the 400 instances used in this article was thus generated independently, which this further complicates direct comparisons. DPLS outperforms all of the three proposed heuristics in [22] consistently on the 6 classes where $v_2 = 0.2$, and the likelyhood of that happening under the assumption that DPLS is no better than the heuristics in [22] is bounded upwards by $0.5^6 = 1.5625\%$. Based on this and the rest of Table 4 we conclude that DPLS is better than the heuristics proposed in [22] on these instances and observe that it is competitive on most of the remaining instances.

On some of the instances with $v_2 = 0.8$ and $m = 20$, the local search performs less impressively, and a closer inspection revealed this to be due to two seperate factors. DPLS performs better when there exists a solution with very little waste, as the search for the optimal solution will terminate faster. Secondly the generation scheme presented in [22] generate bins with an average combined capacity of 9 to 14.5 times the sum of the item sizes on these instances, and as DPLS uses most bins in each subproblem, the running time will suffer considerably. The local search could perform better if the neighborhood was modified to take such instances into account, but the heuristic presented in [11] has already shown promise on such instances.

7.3. Real Life Instances

The instances used in this article are artificially generated to simulate data similar in properties to the largest instances obtained from Danfoam.[12] The number m of different item sizes are varied from two to eight, two being the absolute minimum making sense, and every instance being trivial for the developed heuristics with more than eight item sizes, as all but the last bin typically are filled to optimality by the construction heuristic. The bin sizes are chosen with a uniform distribution between 2900 and 3050 centimeters for 70% of the bins, the rest are choosen from a uniform distribution between 800 and 2900. Enough bins are created to cut out twice the needed amount.

[12]This is done to make comparisons possible by providing generators on request, as the real instances are confidential.

Class	0.25 sec.	0.5 sec.	1 sec.	5 sec.	10 sec.
1	**68.84**	68.54	66.96	61.58	60.81
2	**439.72**	430.22	421.09	413.50	412.18
3	**534.39**	526.29	515.96	505.99	500.37
4	**90.99**	87.78	80.70	71.69	65.68
5	1122.00	907.38	745.29	553.80	511.64
6	1075.65	849.27	732.72	644.49	609.42
7	**31.95**	29.43	27.52	19.95	15.02
8	**295.21**	250.06	218.28	187.79	177.96
9	**413.72**	314.77	270.29	243.19	233.29
10	**39.76**	37.77	35.70	29.45	25.98
11	804.37	502.18	341.48	210.20	183.75
12	963.58	606.29	429.47	308.78	276.30
13	**11.46**	10.50	7.73	3.05	1.73
14	329.54	**202.30**	149.24	96.06	88.78
15	433.12	322.32	**242.04**	150.84	141.48
16	**25.27**	23.97	22.895	15.13	14.36
17	949.96	608.61	386.89	**115.69**	93.00
18	1271.62	830.38	468.21	**138.14**	105.77

Table 4: Results of running DPLS on instances from Table 3 with varying limits on the running time. The bold font marks the earliest result that is better than that of *any* heuristic from [22]

The heuristics performance on the generated instances shows the same characteristics that were observed on the more difficult instances of the real life problem. These instances can thus be used for a fair comparison between heuristics.

The algorithm is set to run for a maximum of ten seconds, but stopping if a zero waste solution is obtained. The average time usage in table 5 thus becomes an indicator for how often that is the case.

It is clear from table 5 that the problem becomes easier as the number of item sizes increases. Further tests indicated that a large number of bin sizes have a positive effect as well. It is also clear that DPLSW usually produces solutions of a better quality than DPLS, though the limited gain on some

n	m	DPC	DPLS	DPLSW	time DPC	time DPLS	time DPLSW
500	2	2186,10	153,40	145,45	0,0078	10,0015	10,0023
500	3	1340,11	9,76	9,43	0,0079	5,1965	5,0155
500	4	1563,57	6,99	1,75	0,0083	1,5303	1,4350
500	5	1538,22	0,28	0,19	0,0087	0,4522	0,4585
500	6	1343,01	0,00	0,00	0,0088	0,0490	0,0388
500	7	1352,97	0,00	0,00	0,0090	0,0523	0,0281
500	8	1392,54	0,00	0,00	0,0090	0,0232	0,0190
1000	2	3157,52	505,97	503,29	0,0288	10,0080	10,0153
1000	3	1910,57	66,03	58,52	0,0311	7,4589	7,7393
1000	4	1763,03	19,84	10,52	0,0406	2,8364	3,2317
1000	5	1554,60	0,74	0,82	0,0329	0,7969	1,0762
1000	6	1474,65	0,00	0,08	0,0344	0,2420	0,4270
1000	7	1446,43	0,09	0,07	0,0351	0,2620	0,2463
1000	8	1346,92	0,00	0,00	0,0359	0,1929	0,1431

Table 5: Average quality of solutions (in terms of units of waste) and time consumption (in seconds) for problems with n items and m different item sizes. Collected over 100 instances

instances might surprise. For the instances with a low number of items, the major part of the waste comes from the inability to fill any bin or most bins and the increased flexibility of ignoring waste on a bin is of limited use. For the instances with a high number of items, the waste is low enough for the randomness of the local searches to play a significant role.

Figure 1 and 2 shows that the local search heuristic usually converges relatively fast, but that it can be prone to random delays as is seen for the run on $n = 1000$ and $m = 8$. This is expected to be a result of optimizations of particularly expensive but irrelevant subproblems, and can be addressed using more finely tuned traversal of the neighborhood, or by traversing it multi threaded. The initial solutions to the local search usually has more than 1000 and sometimes up to 4000 units of waste. To keep the scale in check, this is shown by a line extending beyond 1000 units of waste. If the algorithm ever encounters a solution with no waste, the line will disappear before the tenth second.

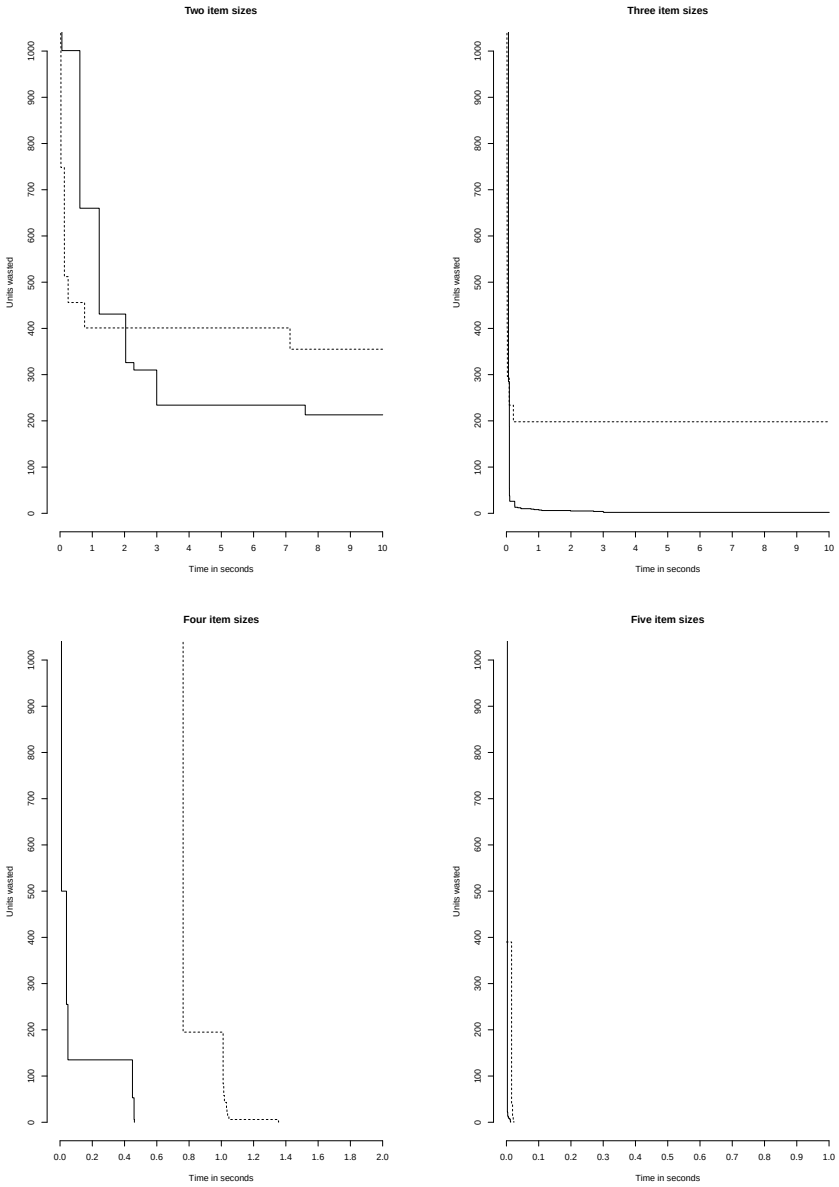

Figure 1: An example of solution quality as a function of time, solid lines and dashed lines being instances with $n = 500$ and $n = 1000$ respectively.

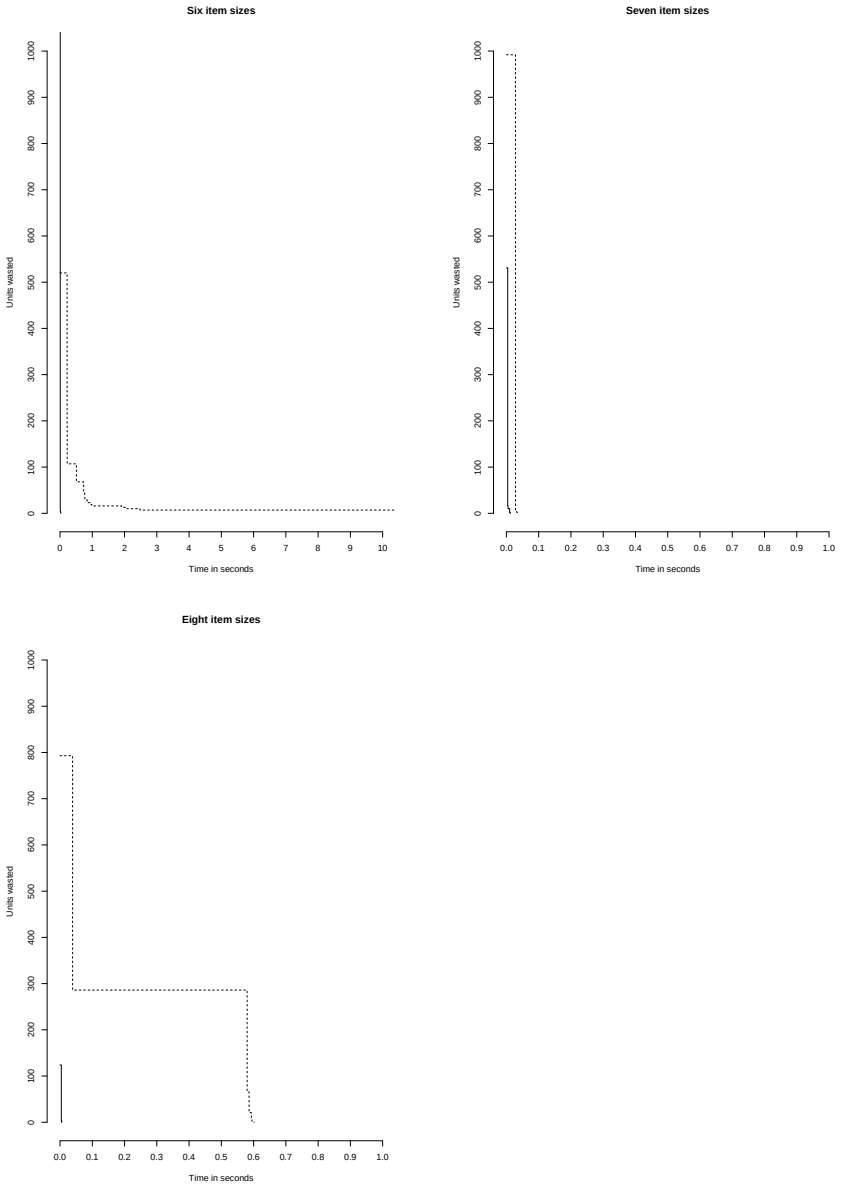

Figure 2: An example of solution quality as a function of time, solid lines and dashed lines being instances with $n = 500$ and $n = 1000$ respectively.

8. Adaptability of the Solver

One of the strengths of the local search heuristic is that it is often flexible in what problems and constraints it is able to handle. Therefore its handling of the following properties should be summed up:

Input minimization vs. output maximization. DPLS is made to handle input minimization problems, but as noted in [3] fairly simple schemes exists for transforming output maximization problems into these.

Objective function. Currently DPLS handles only problems with a one to one correlation between the objective function and the amount of waste on bins, but it has the possibility to deduct a predescribed number of or all usable leftovers.

Homogeneity of items and objects. DPLS generally performs better the more heterogeneous objects and items are, but it has an acceptable performance on problems with homogeneous object and item sizes as well.

Average number of items per object. DPLS currently performs better on problems having a high number of items per object. Some of its deficiencies on instances with a low average number of items per object might be mitigated by further work, but other methods from the literature performs exceedingly well on those.

Time for solving. DPLS can produce reasonably acceptable solutions within half a second on all benchmark instances, and it can potentially use any amount of time available to improve on these until optimality is reached.

The proposed local search heuristic has proven flexible, and as its representation of data is very straightforward, additional constraints that might arise, should often been straightforward to implement.

9. Conclusions

A construction heuristic for the bin packing problem was described, and shown to perform well, also given the larger bin sizes and item to bin ratio than in previous applications of similar methods. Its use is recommended in

all but the very most time or memory critical applications, or instances using *very* large bin sizes.

Methods based on the construction heuristic efficiently solved a real life problem involving very fast machinery, and deadlines measured in milliseconds.

Furthermore a local search heuristic was presented, that scaled well with the number of different bin and item sizes as well as the number of items per bin, at the cost of a poorer scaling with the maximal bin size. The methods also allowed optimization to be resumed at any time, given changes in input, without any ramp up time. Additionally the method allowed for the waste on the worst packed bin to be ignored in a straightforward manner. These properties supplements the methods presented in [6], [7], [11] and [22]. The DPLS heuristic furthermore outperformed the solvers presented in [22] on a large subset of the instances.

The local search heuristic was applied to a real life problem, and improved material usage in Danfoam significantly.

10. Awknowledgements

The authors would like to thank Danfoam and the anonymous company for the cooperation, the real life instances and the feedback provided. The authors are also indebted to the reviewers for their valuable and helpful suggestions, which led to a better presentation and better performance of the heuristic.

[1] P. Gilmore, R. Gomory, A linear programming approach to the cutting-stock problem., Oper. Res. 9 (6) (1961) 849 – 859.

[2] I. Correia, L. Gouveia, F. Saldanha-da Gama, Solving the variable size bin packing problem with discretized formulations, Comput. Oper. Res. 35 (6) (2008) 2103–2113.

[3] G. Wäscher, H. Haußner, H. Schumann, An improved typology of cutting and packing problems, Eur. J. Oper. Res. 183 (3) (2007) 1109 – 1130.

[4] H. Kellerer, U. Pferschy, D. Pisinger, Knapsack Problems, Springer, Berlin, Germany, 2004.

[5] H. Dyckhoff, A typology of cutting and packing problems, Eur. J. Oper. Res. 44 (2) (1990) 145 – 159.

[6] Y. Cui, Y. Yang, A heuristic for the one-dimensional cutting stock problem with usable leftover, Eur. J. Oper. Res. 204 (2) (2010) 245 – 250.

[7] A. C. Cherri, M. N. Arenales, H. H. Yanasse, The one-dimensional cutting stock problem with usable leftover - a heuristic approach, Eur. J. Oper. Res. 196 (3) (2009) 897 – 908.

[8] G. Belov, G. Scheithauer, A cutting plane algorithm for the one-dimensional cutting stock problem with multiple stock lengths, European Journal of Operational Research 141 (2) (2002) 274 – 294.

[9] C. Alves, J. V. de Carvalho, A stabilized branch-and-price-and-cut algorithm for the multiple length cutting stock problem, Comput. Oper. Res.Comput. Oper. Res. 35 (4) (2008) 1315 – 1328.

[10] M. Haouari, M. Serairi, Relaxations and exact solution of the variable sized bin packing problem, Computational Optimization and Applications (2009) 1–24.

[11] M. Haouari, M. Serairi, Heuristics for the variable sized bin-packing problem, Comput. Oper. Res. 36 (10) (2009) 2877–2884.

[12] S. Koch, S. König, G. Wäscher, Integer linear programming for a cutting problem in the wood-processing industry: a case study, International Transactions in Operational Research 16 (6) (2009) 715–726.

[13] S. Dimitriadis, E. Kehris, Cutting stock process optimization in custom door and window manufacturing industry, International Journal of Decision Sciences, Risk and Management 1 (1 – 2) (2009) 66 – 80.

[14] E. Falkenauer, A. Delchambre, A genetic algorithm for bin packing and line balancing., in: Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on, 1992, pp. 1186–1192.

[15] D. Pisinger, Linear time algorithms for knapsack problems with bounded weights, Journal of Algorithms 33 (1) (1999) 1 – 14.

[16] U. Pferschy, Dynamic programming revisited: Improving knapsack algorithms, Computing 63 (1999) 419–430.

[17] K. Fleszar, K. S. Hindi, New heuristics for one-dimensional bin-packing, Comput. Oper. Res. 29 (7) (2002) 821 – 839.

[18] A. Caprara, U. Pferschy, Worst-case analysis of the subset sum algorithm for bin packing, Operations Research Letters 32 (2) (2004) 159 – 166.

[19] A. Caprara, U. Pferschy, Modified subset sum heuristics for bin packing, Information Processing Letters 96 (1) (2005) 18 – 23.

[20] G. Zhang, X. Cai, C. K. Wong, Linear time-approximation algorithms for bin packing, Oper. Res. Lett. 26 (5) (2000) 217 – 222.

[21] J. E. Beasley, Or-library, `http://people.brunel.ac.uk/~mastjjb/jeb/info.html` (July 2009).

[22] K. C. Poldi, M. N. Arenales, Heuristics for the one-dimensional cutting stock problem with limited multiple stock lengths, Comput. Oper. Res. 36 (6) (2009) 2074 – 2081.

[23] D. L. Applegate, L. S. Buriol, B. L. Dillard, D. S. Johnson, P. W. Shor, The cutting-stock approach to bin packing: Theory and experiments, in: Proceedings of the Fifth Workshop on Algorithm Engineering and Experiments, 2002, pp. 1–25.