Fast Pattern-based Algorithms for Cutting Stock

Filipe Brandão

INESC TEC and Faculdade de Ciências, Universidade do Porto, Portugal

fdabrandao@dcc.fc.up.pt

João Pedro Pedroso

INESC TEC and Faculdade de Ciências, Universidade do Porto, Portugal

jpp@fc.up.pt

Technical Report Series: DCC-2013-10

Departamento de Ciência de Computadores

Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 1021/1055,
4169-007 PORTO,
PORTUGAL

Tel: 220 402 900 Fax: 220 402 950

http://www.dcc.fc.up.pt/Pubs/

Fast Pattern-based Algorithms for Cutting Stock

Filipe Brandão
INESC TEC and Faculdade de Ciências, Universidade do Porto, Portugal
fdabrandao@dcc.fc.up.pt

João Pedro Pedroso
INESC TEC and Faculdade de Ciências, Universidade do Porto, Portugal
jpp@fc.up.pt

26 September, 2013

Abstract

The conventional assignment-based first/best fit decreasing algorithms (FFD/BFD) are not polynomial in the cutting stock input size in its most common format. Therefore, even for small instances with large demands, it is difficult to compute FFD/BFD solutions. We present pattern-based methods that overcome the main problems of conventional heuristics in cutting stock problems by representing the solution in a much more compact format. Using our pattern-based heuristics, FFD/BFD solutions for extremely large cutting stock instances, with billions of items, can be found in a very short amount of time.
Keywords: Cutting Stock, First Fit Decreasing, Best Fit Decreasing, FFD, BFD

1 Introduction

The cutting stock problem (CSP) is a combinatorial NP-hard problem (see, e.g., Garey and Johnson 1979) in which pieces of different lengths must be cut from rolls, each with length L, in such a way that the amount of unused space is minimized. In this problem, we are given the length L of the rolls, the number of different piece lengths m, and for each index $k \in \{1, \ldots, m\}$ the corresponding piece length l_k and demand b_k. Since all the rolls have the same length and demands must be met, the objective is equivalent to minimizing the number of rolls that are used.

The bin packing problem (BPP) can be seen as a special case of the cutting stock problem (CSP). In BPP, n objects of m different weights must be packed into a finite number of bins, each with capacity W, in a way that minimizes the number of bins used. However, in the CSP, items of equal size (which are usually ordered in large quantities) are grouped into orders with a required level of demand; in BPP, the demand for a given size is usually close to one. According to Wäscher et al. (2007), cutting stock problems are characterized by a weakly heterogeneous assortment of small items, in contrast with bin packing problems, which are characterized by a strongly heterogeneous assortment of small items.

Since the BPP is hard to solve exactly, many different heuristic solution methods have been proposed. In our paper, we will focus on two heuristics: first fit decreasing (FFD) and best fit decreasing (BFD). Coffman et al. (1997b) and Coffman et al. (1997a) analyze the average and worst case behavior of these two heuristics, and provide an excellent survey on on-line and off-line heuristics for BPP.

The best known implementations of FFD and BFD heuristics run in $\mathcal{O}(n \log n)$ and $\mathcal{O}(n \log(\min(n, W)))$, respectively. Neither of these run times is polynomial in the CSP input size. Therefore, for large CSP instances, computing a FFD/BFD solution may take a very long time using any conventional algorithm.

These algorithms are usually fast enough for BPP, but for large CSP instances better alternatives are necessary.

Johnson (1973) proves that for any implementation of the FFD/BFD heuristics, there exists a list of items of length n for which it will take at least $\Omega(n \log n)$ time to compute a solution. However, his proof requires lists of n items of different weights, which are not usual in cutting stock problems.

Pattern-based algorithms overcome the main problems of conventional approaches in cutting stock problems by representing the solution in a much more compact format. The final solution is represented by a set of patterns with associated multiplicities that indicate the number of times the pattern is used. Moreover, each pattern is represented by the piece lengths it contains and the number of times each of them appears. This representation of the output allows us to avoid the $\Omega(n)$ limit introduced by the assignment piece-by-piece. Besides that, this output format is much more practical in some situations where we are just interested in knowing how to cut the rolls.

The main contributions of this paper are: we present pattern-based algorithms for FFD/BFD heuristics. Our best pattern-based FFD and BFD implementations run in $\mathcal{O}(\min(m^2, n) \log m)$ and $\mathcal{O}(\min(m^2, n) \log(\min(n, W)))$, respectively. Both run times are polynomial in the CSP input size. These algorithms allow us to find solutions for large CSP instances quickly; for example, solutions for instances with one thousand million pieces are found in much less than one second of CPU time on current computers.

The input format of all the algorithms in this paper is the following: m - the number of different lengths; l - the set of lengths; b - the demand for each length; and L - the roll length. The output format depends on the algorithm used. Nevertheless, all the output formats will provide enough information to obtain an assignment piece-by-piece in $\Theta(n)$.

The remainder of this paper is organized as follows. Section 2 presents a straightforward assignment-based FFD algorithm. Section 3 presents the pattern-based FFD algorithm along with an illustrative example. A complexity analysis is presented in Section 3.5. Section 4 and Section 5 present assignment-based and pattern-based BFD algorithms, respectively. Some computational results are presented in Section 6. Finally, Section 7 presents the conclusions.

2 Conventional First Fit Decreasing

The straightforward Algorithm 1 starts by sorting the piece lengths in decreasing order, in line 2, and initializes the solution with a single empty roll, in lines 3-5. The current solution is represented by the number of rolls used (R), the list of assignments (Sol; an assignment of a piece of length l_i to a roll k is represented as $l_i \rightarrow$ roll k) and the list of available spaces (Rem). For each length l_i, the b_{l_i} pieces will be inserted one-by-one in the solution. For each piece, we seek for the first roll where the piece fits. If we find a roll where the piece fits (line 11), we add the piece to the roll and update its remaining space. If there is no roll with enough space (line 17), we create a new one. The variable k' is used to improve the running time in CSP instances by starting to seek for the first roll where the last piece of the same length was inserted (since the length is the same, we know that none of the previous rolls had enough space). The running time of this algorithm is $\mathcal{O}(n + mR)$.

Algorithm 1: Straightforward First Fit Decreasing Algorithm

input : m - number of different lengths; l - set of lengths; b - demand for each length; L - roll length
output: R - number of rolls needed; Sol - list of assignments

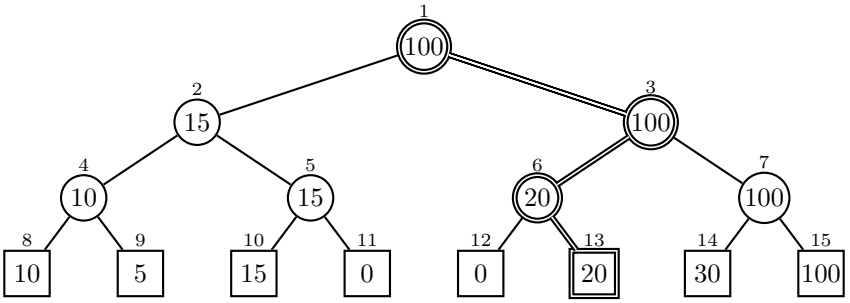
```
1  function FFD(m, l, b, L):
2      l ← reverse(sort(l));                          // sort lengths in decreasing order
3      Sol ← [ ];
4      R ← 1;
5      Rem ← [L];
6      for i ← 1 to m do                              // for each length
7          k' ← 1;
8          for j ← 1 to b_{l_i} do                    // for each piece of length l_i
9              assigned ← False;
10             for k ← k' to R do                     // try each roll
11                 if Rem[k] > l_i then               // if there is enough space
12                     Rem[k] ← Rem[k] − l_i;
13                     Sol.append(l_i → roll k);
14                     assigned ← True;
15                     k' ← k;
16                     break;
17             if not assigned then                   // if the piece was not assigned to any roll
18                 R ← R + 1;
19                 k' ← R;
20                 Rem.append(L − l_i);
21                 Sol.append(l_i → roll R);
22     return (R, Sol);
```

2.1 Efficient implementation

Johnson (1973) shows how to implement the FFD algorithm to run in $\mathcal{O}(n \log n)$ time, where n is the number of objects, using a tree. His tree was generalized for other applications and nowadays it is known as tournament tree, or as winner tree. A winner tree is a complete binary tree in which each node represents the best of its two children. For BPP, we need a tree with at least n leaves corresponding to bins. The value of each leaf is the remaining available space. Figure 1 shows a winner tree applied to a small BPP instance. To find the leftmost bin with at least a certain gap g, we merely start at the root node and traverse to a leaf by choosing always the leftmost child that leads to a node with gap $\geq g$. After the insertion of the item, we update the label of the leaf node and then proceed backing up the path to the root, relabeling each node if necessary. Since the tree has height at most $\lceil \log_2 n \rceil$, it takes $\mathcal{O}(\log n)$ time to identify the bin and $\mathcal{O}(\log n)$ time to update the tree. Therefore, this operation of insertion of an item takes $\mathcal{O}(\log n)$ time. Since we are dealing with a complete binary tree, we can represent this data structure efficiently using a simple array.

Algorithm 2 shows a possible implementation of a winner tree for the FFD heuristics. For the sake of simplicity, we round the number of leaves to the smallest power of two greater than or equal to n. The procedure `initWTree` initializes the winner tree; `winner`(g) finds the leftmost roll with gap at least g; and `update`(j, g) updates the available space of the roll j to the value g.

Figure 1: Winner tree.



Winner tree before an assignment of a piece of length 18 when the first 7 rolls, of capacity 100, are filled with 90, 95, 85, 100, 100, 80 and 70, respectively. The highlighted path indicates the search path along the tree that finds the leftmost bin with gap at least 18. The number inside each node is the key value and the number above is the index in the array representation.

Algorithm 2: Winner tree implementation for the First Fit Decreasing heuristics

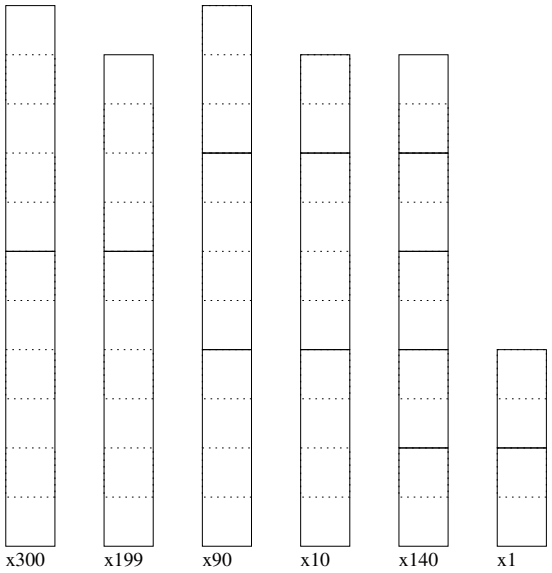
```
1  function initWTree(n, W):                          // initializes the tree
2      n' ← 2^⌈log2 n⌉;                               // n' is the smallest power of 2 greater than or equal to n
3      size ← 2n' − 1;                                // number of nodes in the tree
4      key[i] ← W, for 1 ≤ i ≤ size;                  // initialize all the nodes with W
5  function winner(g):                                // finds the leftmost leaf with value at least g
6      i ← 1;
7      while 2i ≤ size do
8          if key[2i] ≥ g then i ← 2i;                // try the left child first
9          else i ← 2i + 1;
10     return i − n' + 1;                             // return the index of the leaf
11 function update(j, g):                             // updates the j-th leaf with value g
12     i ← size − n' + j;                             // compute the array-index of the j-th leaf
13     key[i] ← g;                                    // update the leaf
14     while i > 1 do                                 // propagate the update
15         p ← i/2;
16         t ← max(key[2p], key[2p + 1]);
17         if key[p] = t then break;
18         key[p] ← t;
19         i ← p;
```

3 Pattern-Based First Fit Decreasing

To represent pattern-based solutions, we will use a set of pairs (m_i, p_i) where the multiplicity m_i is the number of times the pattern p_i appears in the solution, and each pattern p_i is a list $[a_1 \times (l = l_1), a_2 \times (l = l_2), ...]$ where each $a_j \times (l = l_j)$ represents an item of length l_j repeated a_j times.

Figure 2 shows a FFD solution for an instance with $L = 11$, and pieces of lengths 6, 5, 4, 3, 2 with demands 499, 300, 399, 90, 712, respectively; hence, $n = 2000$ and $m = 5$. Despite having 2000 items, the solution can be represented compactly in the pattern format as follows: $\{(300, [1 \times (l = 6), 1 \times (l = 5)]), (199, [1 \times (l = 6), 1 \times (l = 4)]), (90, [2 \times (l = 4), 1 \times (l = 3)]), (10, [2 \times (l = 4), 1 \times (l = 2)]), (140, [5 \times (l = 2)]), (1, [2 \times (l = 2)])\}$.

Figure 2: An example of a first fit decreasing solution.



First fit decreasing solution for an instance with $L = 11$ and pieces of lengths 6, 5, 4, 3, 2 with demands 499, 300, 399, 90, 712, respectively; Despite having 2000 items, there are only 6 different patterns in the solution.

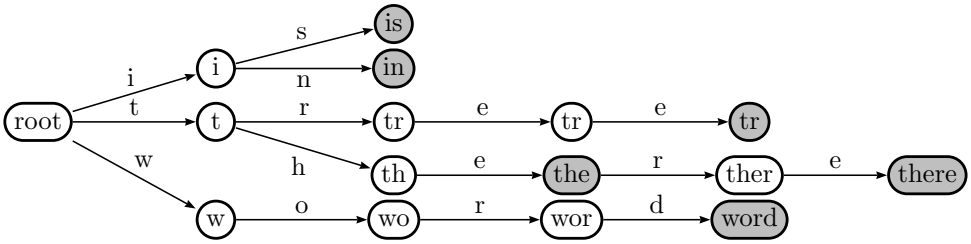
3.1 Prefix tree

A trie, also called a prefix tree, is a tree structure that is usually used to store words. Our algorithm uses a variant of this data structure to represent patterns and their multiplicities. In this section, we will explain how to interpret this kind of trees.

Figure 3 shows a small prefix tree containing the words {"there", "the", "word", "is", "in", "tree"}. Every path from the root to a node represents a prefix of words stored in the tree. The gray nodes are called terminal nodes. Paths from the root to terminal nodes correspond to words stored in the tree. The list of words can be retrieved by storing every path that ends in a terminal node while traversing the tree.

In the same way that words are stored in a trie, we can store patterns of the form $[a_1 \times (l = l_1), a_2 \times (l = l_2), ...]$ considering each $a_i \times (l = l_i)$ as a symbol. However, we also need to know their multiplicities; but it is easy to modify the trie in order to count the number of occurrences of each prefix. Figure 4 shows a prefix tree with prefix count. The words from the list ["there", "the", "the", "word", "is", "in", "tree'] are stored in this tree. Every word appears once except the word "the" that appears twice. The ρ value of an arc can be seen as the number of words that use the arc. The number of times a prefix occurs is given by the ρ value of its last arc. The number of times a prefix appears in the original list is given by

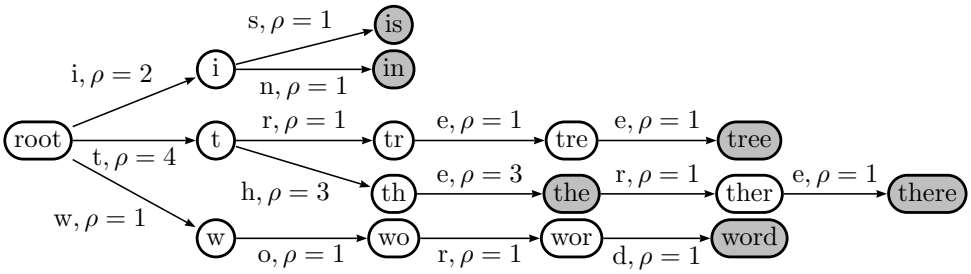
Figure 3: A simple prefix tree.



A simple prefix tree containing the words: {"there", "the", "word", "is", "in", "tree"}. Every path from the root to a node represents a prefix of the words stored in the tree. The gray nodes are called terminal nodes and contain the stored words.

the difference between the ρ value of the arc from its parent and the sum of the ρ values of arcs to its children. If this difference is greater than zero, the word exists by itself in the original list; otherwise, it is just a prefix of other words.

Figure 4: A prefix tree with prefix count.



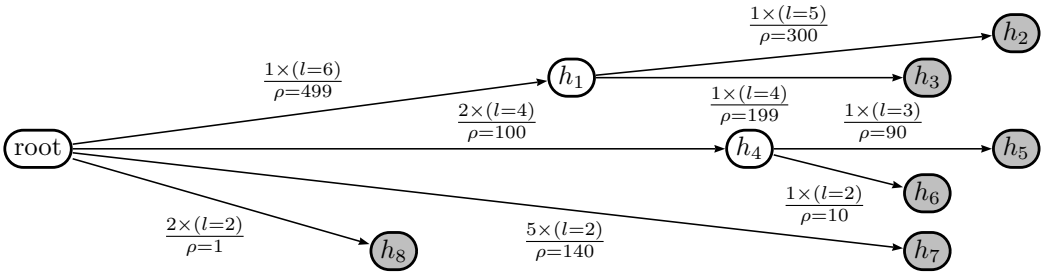
A prefix tree containing the prefixes and prefix occurrence counts (ρ) of the words in the list ["there", "the", "the", "word", "is", "in", "tree"]. The ρ value of an arc can be seen as the number of words that use the arc.

With the introduction of the prefix count, we can now store solutions, in the previously described format, in a trie. Figure 5 shows the solution from Figure 2 stored in a trie. Every path from the root to a node represents a sub-pattern and the ρ of the last arc in the path indicates its multiplicity. The gray nodes are terminal nodes corresponding to patterns in the solution. The list of patterns and their multiplicities can be retrieved by storing every path that ends in a terminal node, along with the ρ of its last arc, while traversing the tree. We do not store patterns in the nodes, since we can always obtain them from the paths.

3.2 Algorithm

In the pattern-based Algorithm 3, the solution is represented by a tree as described in the previous section. The tree is represented by the following data structures: $\mathsf{Label}[u, v]$ indicates which piece appears between nodes u and v and the number of times it appears using the notation $a_i \times (l = l_i)$; $\mathsf{Count}[u, v]$, which corresponds to the ρ value of the arc, indicates how many times the arc is used. Each vertex in the graph is identified by two values: h and s. The value s indicates the amount of roll used by the corresponding sub-pattern and h is just an identifier to distinguish between nodes with the same value s. The solution is initialized with an empty tree, in lines 2-3. The piece lengths are sorted in decreasing order, in line 4. The variable h' is just the value h of the last node created. In this algorithm we only modify patterns that end in terminal nodes, since non-terminal nodes are just sub-patterns of other patterns. Q^* is a list of terminal nodes and their multiplicities, initialized with the root node $(h = 0, s = 0)$ with multiplicity ∞; this means that the empty pattern (root) is allowed to appear an

Figure 5: Tree representation of the first fit decreasing solution from Figure 2.



Every path from the root to a node represents a sub-pattern and the ρ value of the last arc in the path indicates its multiplicity. Sub-patterns ending in terminal nodes are patterns that appear by itself in the solution. Paths that end in non-terminal nodes are just sub-patterns of patterns in the solution. The root node represents the empty pattern.

infinite number of times in the solution and hence it is always possible to start new patterns from the empty pattern. Recall that the multiplicity of every node except the root will be equal to the difference between the ρ value of the incident arc and the sum of the ρ values of outgoing arcs, as described in the previous section.

In the main loop of line 8, for each length l_i we insert the b_{l_i} pieces in the solution by creating new patterns starting from the current terminal nodes. Q is the list of terminal nodes for the following iteration, and it is initialized empty, in line 9. In the loop of line 11, we test each terminal node in the order they appear in Q^*, checking if there is available space to introduce some pieces with the current length l_i. If there is available space, the value γ (line 13) will be larger than 0 and will indicate the maximum number of times the current piece fits in the terminal node (**div** denotes integer division and **mod** is its remainder). If γ is larger than 0 (line 14), the value ρ' (line 16) will indicate the number of rolls that can include the current piece the maximum number of times it fits. If ρ' is zero, no new pattern can be added; otherwise (line 17), the multiplicity of the current terminal node is decreased and a new pattern is created by adding a new terminal node. Then, in line 23, we check if there are remaining pieces of the current length and space to insert them in the current position. If this is the case, another pattern is formed by adding a new terminal node, and the multiplicity of the current node is decreased. In line 28, we check if the current node remains as a terminal node. This process is repeated for every length i and finally, in line 31, we compute the number of rolls needed by summing the multiplicities of the terminal nodes except the root. This algorithm runs in $\mathcal{O}(m^2)$ time; see Section 3.5 for more details.

Algorithm 4 receives the output of Algorithm 3 and produces an output in the format described in the beginning of Section 3. This algorithm uses a simple recursive function that performs a depth-first traversal of the tree and whenever it finds a terminal node it adds the corresponding pattern to the solution (lines 3-10). In line 2, an adjacency list for the tree is obtained. The recursive function is called, in line 11, starting at the node $(h = 0, s = 0)$ with an empty pattern with multiplicity R. In line 6, $\mathsf{path} + \mathsf{Label}[u, v]$ stands for the result of appending $\mathsf{Label}[u, v]$ to the list path. This algorithm is linear in the length of the output, which is limited by $\mathcal{O}(\min(m^2, n))$; see Section 3.5 for more details.

Algorithm 3: Pattern-Based First Fit Decreasing Algorithm

input : m - number of different lengths; l - set of lengths; b - demand for each length; L - roll length
output: R - number of rolls needed; Count - ρ value of each arc; Label - length of the piece associated with each arc and the number of times it is repeated

function `FFDTree`(m, l, b, L):
  Count$[u, v] \leftarrow 0$, **for all** u, v; *// initialize auxiliary data structures*
  Label$[u, v] \leftarrow$ NIL, **for all** u, v;
  $l \leftarrow$ `reverse`(`sort`(l)); *// sort lengths in decreasing order*
  $h' \leftarrow 0$; *// index of the last node*
  root $\leftarrow ((h', 0), \infty)$; *// empty pattern*
  $Q^* \leftarrow$ [root]; *// list of terminal nodes*
  for $i \leftarrow 1$ **to** m **do** *// for each piece length*
    $Q \leftarrow [\,]$;
    $\beta \leftarrow b_{l_i}$;
    foreach $((h, s), r) \in Q^*$ **do** *// for each terminal node in the order they appear in Q^**
      if $\beta > 0$ **then**
        $\gamma \leftarrow (L - s)$ **div** l_i;
        if $\gamma > 0$ **then**
          $u \leftarrow (h, s)$;
          $\rho' = \min(r, \beta$ **div** $\gamma)$;
          if $\rho' > 0$ **then** *// if there is remaining demand to use the piece the maximum number of times it fits*
            $\beta \leftarrow \beta - \gamma\rho'$; $r \leftarrow r - \rho'$;
            $h' \leftarrow h' + 1$; $v \leftarrow (h', s + \gamma l_i)$;
            Count$[u, v] \leftarrow \rho'$; Label$[u, v] \leftarrow (\gamma \times l_i)$;
            Q.append$((v,\ \rho'))$;
          $\gamma' \leftarrow \beta$ **mod** γ;
          if $r > 0$ **and** $\gamma' > 0$ **then** *// if there remains demand and multiplicity to insert the remaining pieces*
            $\beta \leftarrow \beta - \gamma'$; $r \leftarrow r - 1$;
            $h' \leftarrow h' + 1$; $v \leftarrow (h', s + \gamma' l_i)$;
            Count$[u, v] \leftarrow 1$; Label$[u, v] \leftarrow (\gamma' \times l_i)$;
            Q.append$((v,\ 1))$;
      if $r > 0$ **then** *// check if the current node remains as a terminal node*
        Q.append$(((h,\ s),r))$;
    $Q^* \leftarrow Q$;
  R $\leftarrow \sum_{((h,s),r) \in Q^* \setminus \{\text{root}\}} r$;
  return (R, Count, Label);

Algorithm 4: Solution Extraction Algorithm

input : R - number of rolls needed; Count - ρ value of each arc; Label - length of the piece associated with each arc and the number of times it is repeated

output: Set of patterns and multiplicities

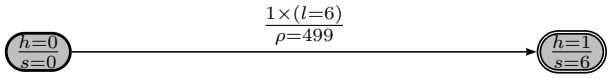
```
1  function patternExtraction(R, Count, Label):
2      Adj ← getAdjList(Label);                          // obtain an adjacency list of the tree
3      function getPatterns(u, ρ', path):                // recursive function to extract patterns
4          patterns ← { };
5          foreach v ∈ Adj[u] do
6              patterns ← patterns ∪ getPatterns(v, Count[u, v], path + Label[u, v]);
7              ρ' ← ρ' − Count[u, v];
8          if ρ' > 0 then                                // if it is a terminal node, add the pattern to the solution
9              patterns ← patterns ∪ {(ρ', path)};
10         return patterns;
11     return getPatterns((0, 0), R, [ ]);
```

3.3 Example

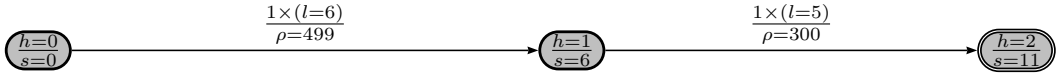
Consider an instance with $L = 11$ and pieces of lengths 6, 5, 4, 3, 2 with demands 499, 300, 399, 90, 712, respectively; hence, $n = 2000$ and $m = 5$.

- We start with a tree with only one node corresponding to the empty pattern. This pattern has infinite multiplicity, which means that there is no limit on the number of patterns that can be created starting from it.

- After the insertion of the 499 pieces of length 6, we have a tree with two nodes; one corresponding to the empty pattern [ ] and another corresponding to the pattern $[1 \times (l = 6)]$. Both are terminal nodes since their multiplicity is greater than zero. Recall that the multiplicity of every node except the root is equal to the difference between the ρ of the arc from its parent and the sum of the ρ values of arcs to its children. The node $(h = 0, s = 0)$ remains with infinite multiplicity, and the node $(h = 1, s = 6)$ has multiplicity 499, meaning that it is possible to create more patterns starting from it by adding more pieces.

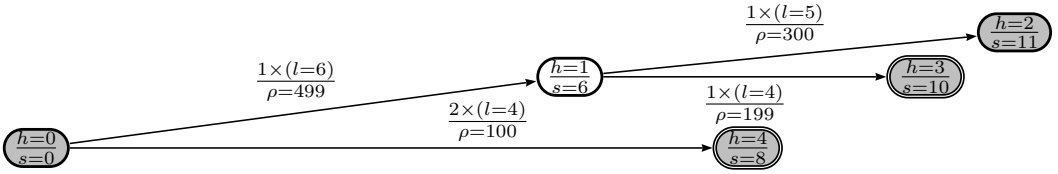


- At the third iteration, we insert the pieces of length 5. The first terminal node in the list Q^* with available space is the node $(h = 1, s = 6)$. This node has multiplicity 499, allowing it to hold all the pieces with $l = 5$, so we just add a child node. The tree is now composed by three terminal nodes. The solution represented by this tree is $\{(300, [1 \times (l = 6), 1 \times (l = 5)]), (199, [1 \times (l = 6)])\}$, i.e., there are 300 rolls with two pieces, one piece of length 6 and another of length 5, and 199 rolls with one piece of length 6. Note that the node $(h = 1, s = 6)$ remains as a terminal node but its multiplicity decreased to 199, since a pattern that uses 300 units of its multiplicity has been created starting from it.

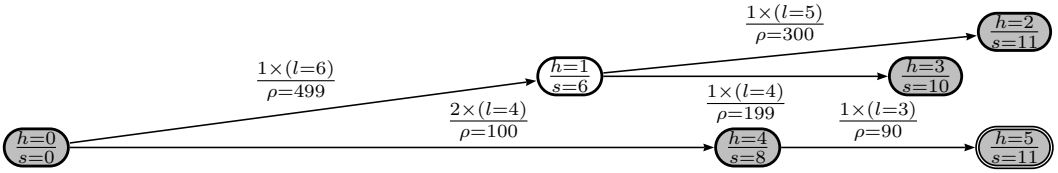


- After the insertion of the 399 pieces of length 4, the node $(h = 1, s = 6)$ is no longer a terminal node since the pattern $(199, [1 \times (l = 6), 1 \times (l = 4)])\}$ used the remaining multiplicity. The first terminal node with available space was $(h = 1, s = 6)$ and it was used to hold 199 pieces. For the remaining 100 pieces of length $l = 4$, a new pattern starting with two pieces of length 4 has been formed, since

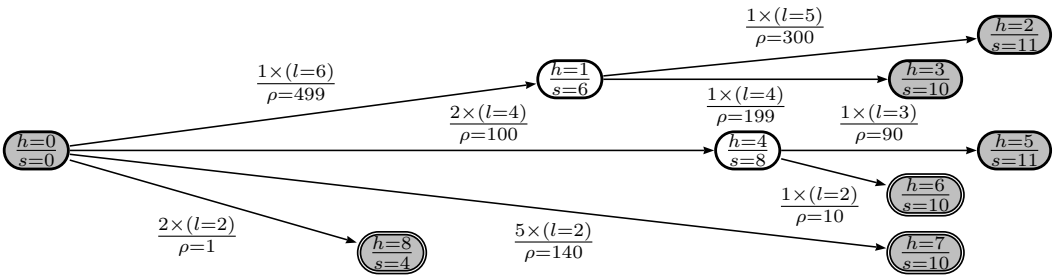
the next terminal node with available space was $(h = 0, s = 0)$ and 11 **div** 4 = 2. The current solution represented by this tree is $\{(300, [1 \times (l = 6), 1 \times (l = 5)]), (199, [1 \times (l = 6), 1 \times (l = 4)]), (100, [2 \times (l = 4)])\}$.



- At the fifth iteration, we add the pieces of length 3. The first terminal node in Q^* with available space is the node $(h = 4, s = 8)$. Since the multiplicity of this node is 100 and we have 90 pieces of length 3, we just add a new pattern $[2 \times (l = 4), 1 \times (l = 3)]$ by adding a child node to $(h = 4, s = 8)$.



- Now, to add the 712 pieces of length 2, things are a little more complicated. The first terminal node in Q^* with available space is the node $(h = 4, s = 8)$. The multiplicity of this node is 10, so we replace the pattern $[2 \times (l = 4)]$ by $[2 \times (l = 4), 1 \times (l = 2)]$ adding a child node $(h = 6, s = 10)$ that uses the remaining multiplicity. Now, there remain 702 pieces to add. Since there are more pieces to add, we proceed to the next terminal node, which is $(h = 0, s = 0)$. Starting from this node, it is possible to use 140 times the pattern $[5 \times (l = 2)]$ since 11 **div** 2 = 5 and 702 **div** 5 = 140. Since, we still have two items of length 2 to add and the multiplicity of the node $(h = 0, s = 0)$ is infinite, we just add the pattern $[2 \times (l = 2)]$.



- The final solution represented by this tree is $\{(300, [1 \times (l = 6), 1 \times (l = 5)]), (199, [1 \times (l = 6), 1 \times (l = 4)]), (90, [2 \times (l = 4), 1 \times (l = 3)]), (10, [2 \times (l = 4), 1 \times (l = 2)]), (140, [5 \times (l = 2)]), (1, [2 \times (l = 2)])\}$ and it uses 6 different patterns and 740 rolls.

3.4 Efficient implementation

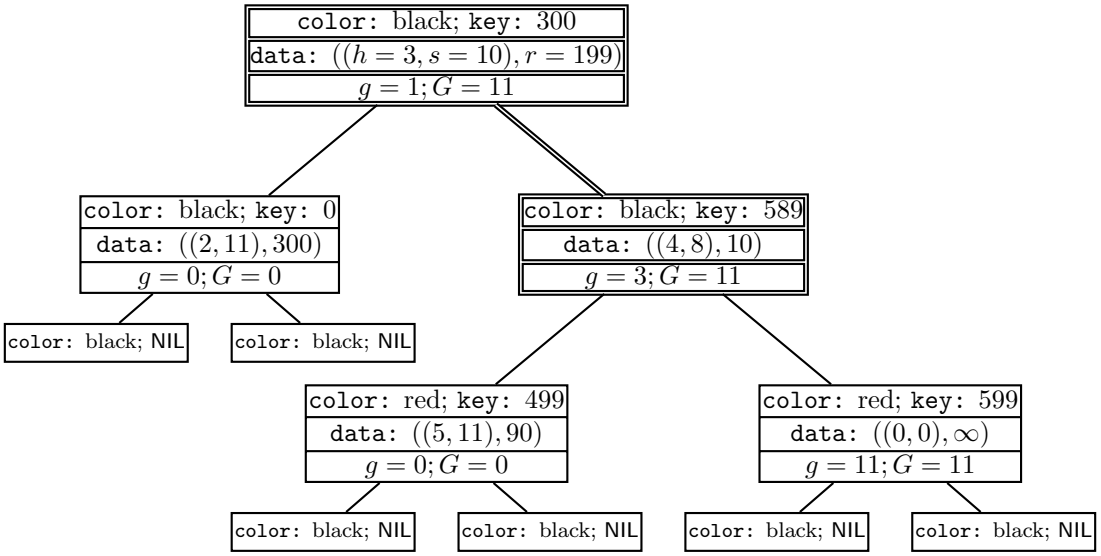
The pattern-based algorithm presented in the previous section runs in $\mathcal{O}(m^2)$ time and it is much faster than any conventional algorithm in CSP instances. However, in BPP instances, when $m = n$, it is quadratic in the number of items. To overcome this problem, we introduce a new data structure that we call red-black winner tree, since it results from the combination of these two data structures. This data structure will allow us to compute any FFD solution in $\mathcal{O}(\min(m^2, n) \log m)$ time, which is at least as fast as the most efficient conventional FFD implementation in BPP instances and extremely fast in CSP instances.

A red-black tree is a self-balancing binary search tree that is typically used to implement associative arrays. Since it is a balanced tree, it guarantees insertion, search and delete to be logarithmic in the number of elements in the tree. This type of tree was initially proposed by Bayer (1972) with the name

symmetric binary B-tree. In this data structure, each node has a color attribute, which is either red or black, and there is a set of rules that need to be always satisfied: every node is either red or black; the root is black; the leaves are sentinel nodes with no data, and their color is black; every node has two children (left child with a smaller key and right child with a larger key); both children of every red node are black; every simple path from a given node to any of its descendant leaves contains the same number of black nodes. It can be proved that these constraints guarantee that the height of the tree is no longer than $2\log(n+1)$, where n is the total number of elements stored in the tree (see, e.g., Cormen et al. 2001).

The red-black winner tree is a self-balancing search tree in which each node contains a key, a data field, a value (g), and G, the maximum value in the subtree below the node including the node value. The key is used to keep the patterns sorted by position in a virtual list of rolls (i.e., by the index of the first roll that uses the pattern), the data field is associated with a node in the graph representating the FFD solution and the value is simply the gap of the pattern represented by the node. Figure 6 shows a small red-black winner tree applied to the example for the pattern-based FFD algorithm before the assignment of the pieces of length 2.

Figure 6: Red-black winner tree.



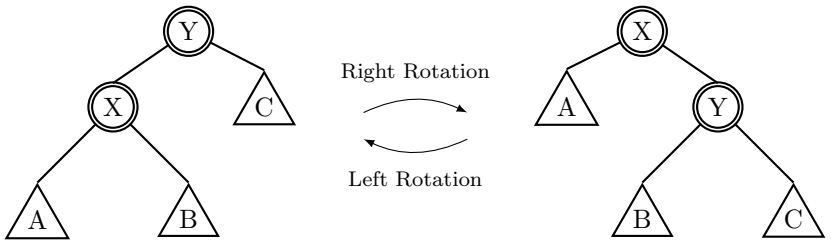
Red-black winner tree corresponding to the example for the pattern-based FFD algorithm before the assignment of the pieces of length 2. The highlighted path indicates the search path along the tree that finds the leftmost pattern with gap at least 2. Note that each pattern is represented by its corresponding terminal node in the pattern prefix tree. The key is the index of the first roll that uses the pattern stored in the node. The winner is the node with `key` $= 589$, and the corresponding node in the pattern tree is $(h = 4, s = 8)$, which has multiplicity 10. The insertion of the pieces of length 2 in this pattern modifies its data field and its g value. In this particular situation, the G value remains unchanged and we do not need to proceed back up the tree updating the G values; these updates stop at the first node whose G value remains unchanged since it is the G value that is propagated. The leaves are the sentinel nodes.

A standard red-black tree only has in each node a key and a data field. We need to modify slightly some of the operations because of the G field. More specifically, we need to modify the update, insertion and the rotations. Moreover, we need to add a winner operation. As our algorithm only requires insertions, we will not specify the details about a modified delete operation.

When inserting a new node in a red-black tree, we start at the root and follow one of the child nodes depending on the key value. We stop when we find a leaf that will be replaced by the new node. When inserting a node with gap g' in a red-black winner tree, on the way down, we need to update G to $\max(G, g')$ in every node along the path to the leaf where the node will initially be inserted. After the insertion of the node, it is colored red and two black leafs are added as children. Depending on the color of other nearby nodes it may be necessary to recolor nodes and/or apply rotations along the path up to the root in order to maintain the invariants, see, e.g., Cormen et al. (2001) for more details. Rotations

interfere with the G field since it is the maximum g value in the subtree below a node. Therefore, we need to modify this operation too. Figure 7 shows what happens during each type of rotation. None of the rotations will change the G values inside A, B and C. Therefore, we just need to recompute the G values of the X and Y nodes.

Figure 7: Red-black tree rotations.



After rotations in red-black winner trees, we just need to recompute the G values of the X and Y nodes, since none of the rotations will change the G values inside A, B and C.

Algorithm 5 shows how to implement the update and the winner operations. The operation `winner`(g) finds the leftmost pattern with gap at least g; and `update`$(\mathsf{key}, \mathsf{data}, g)$ replaces the pattern at position key by a new pattern data with gap g. Recall that each pattern is represented by its corresponding terminal node in the pattern prefix tree. When we update the data field of a node, we do not need to do anything. However, when we update the g value, we need to propagate it along the path to the root. The `winner` operation is somewhat similar to the same operation in a normal winner tree: we just look for the node with the smallest key and enough available space, since the key is the index of the first roll that uses the pattern stored in the node.

Algorithm 5: Red-black winner tree update and winner operations

```
1  function winner(g):                                   // finds the node with the smallest key and value at least g
2      x ← root;
3      while True do
4          if x.left ≠ NIL and x.left.G ≥ g then x ← x.left;        // try the left subtree
5          else if x.g ≥ g then break;                              // try the current node
6          else x ← x.right;                                        // try the right subtree
7      return x;
8  function update(key, data, g):                        // updates the value and the data field of a node
9      x ← search(key);
10     x.g ← g;
11     x.data ← data;
12     while True do
13         t ← max(x.left.G, x.right.G, x.g);
14         if x.G = t or x = root then break;           // stop at the root or when the G value remains unchanged
15         x.G ← t;
16         x ← x.parent;
```

Algorithm 6 shows how to use the red-black winner tree in conjunction with Algorithm 3. This algorithm is very similar to the Algorithm 3, the main difference is the fact that we store the patterns in a red-black winner tree instead of using lists. For each length, while there remains demand, we use the winner operation to find the leftmost pattern with enough available space. The winner pattern will be replaced by a new pattern; another new pattern may appear, and if there remains multiplicity in the original pattern, it will be inserted again in the tree with the remaining multiplicity. We store in a list the new patterns that appear; we update the winner with the first pattern; and if there are other patterns in the list, we insert them in the tree.

Algorithm 6: Efficient Pattern-Based First Fit Decreasing Algorithm

input : m - number of different lengths; l - set of lengths; b - demand for each length; L - roll length
output: R - number of rolls needed; Count - ρ value of each arc; Label - length of the piece associated with each arc and the number of times it is repeated

```
1  function FFDTree(m, l, b, L):
2      Count[u, v] ← 0, for all u, v ;                          // initialize auxiliary data structures
3      Label[u, v] ← NIL, for all u, v;
4      l ← reverse(sort(l));                                    // sort lengths in decreasing order
5      h' ← 0; root ← ((h', 0), ∞);
6      T ← RBWT([(key = 0, data = root, g = L)]);               // initialize the red-black winner tree with the root
7      for i ← 1 to m do                                        // for each piece length
8          β ← b_{l_i};
9          while β > 0 do                                       // while there remains demand for length i
10             N ← [ ];
11             key ← T.winner (l_i);
12             ((h, s), r) ← T[key];
13             γ ← (L − s) div l_i;
14             if γ > 0 then
15                 u ← (h, s);
16                 ρ' = min(r, β div γ);
17                 if ρ' > 0 then                               // if there is remaining demand to use the piece the maximum number of times it fits
18                     β ← β − γρ'; r ← r − ρ';
19                     h' ← h' + 1; v ← (h', s + γ l_i);
20                     Count[u, v] ← ρ'; Label[u, v] ← (γ × l_i);
21                     N.append(((v, ρ'), L − v_2));
22                 γ' ← β mod γ;
23                 if r > 0 and γ' > 0 then                     // if there remains demand and multiplicity to insert the remaining pieces
24                     β ← β − γ'; r ← r − 1;
25                     h' ← h' + 1; v ← (h', s + γ' l_i);
26                     Count[u, v] ← 1; Label[u, v] ← (γ' × l_i);
27                     N.append(((v, 1), L − v_2));
28             if r > 0 then                                    // if there remains multiplicity for the original pattern
29                 N.append((((h, s), r), L − s));
30             (v, f), g ← N.pop(0);
31             T.update (key, (v, r), g);
32             foreach ((v, r), g) ∈ N do
33                 T.insert (key, (v, r), g);
34                 key ← key + r;
35     R ← T.max();                                             // the maximum key value corresponds to the position of the empty pattern
36     return (R, Count, Label);
```

3.5 Complexity analysis

Theorem 1 *The number of different patterns p_m in a FFD/BFD solution is at most $2m$, where m is the number of different piece lengths.*

Proof The proof is made by induction on m.

- The case $m = 1$ is verified as follows: using a single length we have at most two patterns, one composed by the maximum number of times that the pieces fit the roll and another composed by the remaining pieces.
- Assume $p_m \leq 2m$ holds. There are three (non-exclusive) situations that can happen when introducing a new piece length:
  1. replace entirely some patterns by new patterns that include the current length (the total number of patterns remains the same);
  2. modify some pattern by reducing its multiplicity and adding:
     - one pattern where the piece is used the maximum number of times it fits the roll;
     - and/or, one pattern with multiplicity one with the remaining pieces.

  By the induction hypothesis, $p_m \leq 2m$; since we add at most two patterns by introducing a new piece length, with $m + 1$ piece lengths we will have $p_{m+1} \leq 2m + 2 = 2(m + 1)$ patterns. This result is valid for FFD and BFD solutions.

□

The pattern-based FFD Algorithm 3 has complexity $\mathcal{O}(mp)$, where m is the number of different piece lengths and p is the number of different patterns in the final solution. Theorem 1 ensures that the number of patterns will be at most $2m$. The complexity of the Algorithm 3 is therefore $\mathcal{O}(m^2)$. For BPP instances we still have $\mathcal{O}(n^2)$ if all the pieces have different lengths, i.e., $m = n$, but this algorithm is intended to CSP instances where the number of different lengths is usually much smaller than the total number of pieces. Also, this algorithm is polynomial in the CSP input size, as opposed to a conventional $\mathcal{O}(n \log n)$ FFD algorithm which is pseudo-polynomial.

Let us consider each $a_i \times (l = l_i)$ (i.e., piece of length l_i repeated a_i times) as a symbol. Let χ be the total number of symbols in the output and k the average number of symbols per pattern (i.e., $k = \chi/p$). The number of patterns p is limited by $2m$, and k is usually very small in FFD/BFD solutions when the demands are high, since each length is used as many times as possible leaving little space for other pieces. The pattern extraction algorithm, Algorithm 4, runs in $\mathcal{O}(\min(m^2, n))$ time, since it is linear in the length of the output (i.e., in the number of symbols). More precisely, it runs in $\mathcal{O}(kp)$. An assignment piece-by-piece can be obtained from the pattern representation in $\Theta(n)$.

Algorithm 6 runs in $\mathcal{O}(\min(m^2, n) \log m)$ time since it performs $\mathcal{O}(\log m)$ operations per arc. More precisely, since the number of arcs is no more than the number of symbols in the output, this algorithm runs in $\mathcal{O}(kp \log m)$, where p is limited by $2m$ and k is usually very small, in cutting stock instances. Therefore, this algorithm is as fast as the most efficient known implementation of the FFD heuristics in BPP instances, and it is extremely fast in cutting stock instances.

4 Conventional Best Fit Decreasing

The BFD Algorithm 7 starts by sorting the piece lengths in decreasing order (line 2) and it initializes the solution with a single empty roll (lines 3-5). The current solution is represented by the number of rolls used (R), the list of assignments (Sol; an assignment of a piece of length l_i to a roll k is represented as $l_i \rightarrow$ roll k) and the list of the available spaces (Rem). For each length l_i, the b_{l_i} pieces will be inserted one-by-one in the solution. For each piece, we seek the roll with smallest gap greater or equal to the piece length. If we find rolls where the piece fits (line 9), we add the piece to one with the smallest gap and we update its remaining space. If there is no roll with enough space (line 13), we create a new one.

The naive implementation of this algorithm runs in $\mathcal{O}(n \min(n, L))$ where n is the number of pieces and L the length of the rolls, since it takes $\mathcal{O}(\min(n, L))$ time the identify the "best" roll. However, it can be implemented to run in $\mathcal{O}(n \log(\min(n, L)))$ time using a self-balancing search tree such as a red-black tree.

Algorithm 7: Straightforward Best Fit Decreasing Algorithm

input : m - number of different lengths; l - set of lengths; b - demand for each length; L - roll length
output: R - number of rolls needed; Sol - list of assignments

```
1  function BFD(m, l, b, L):
2      l ← reverse(sort(l));                          // sort lengths in decreasing order
3      Sol ← [ ];
4      R ← 1;
5      Rem ← [L];
6      for i ← 1 to m do                              // for each length
7          for j ← 1 to b_{l_i} do                    // for each piece of length l_i
8              S ← {k | 1 ≤ k ≤ R, Rem[k] ≥ l_i};
9              if S ≠ ∅ then
10                 best ← argmin_{k∈S} Rem[k];
11                 Rem[best] ← Rem[best] − l_i;
12                 Sol.append(l_i → roll best);
13             else                                   // if there is no roll with enough available space
14                 R ← R + 1;
15                 Rem.append(L − l_i);
16                 Sol.append(l_i → roll R);
17     return (R, Sol);
```

4.1 Efficient implementation

Algorithm 8 presents a straightforward implementation of upper bound operation that finds the smallest key greater than or equal to a certain value in a binary search tree. This operation allows us to find the smallest gap sufficient for the item size such that at least one roll has such gap, or an indication that no such roll exists, in $\mathcal{O}(\log(\min(n, L)))$ time when using a self-balancing binary search tree. By keeping the rolls grouped by gap in a tree, such as a red-black tree, the n items will be handled in $\mathcal{O}(n \log(\min(n, L)))$.

Algorithm 8: Upper Bound Operation

```
1  function upperBound(value):
2      x ← root;
3      last ← NIL;
4      while x ≠ NIL do
5          if x.key = value then return x.key ;
6          else if x.key > value then
7              last ← x.key;
8              x ← x.left;
9          else
10             x ← x.right;
11     return last;
```

5 Pattern-Based Best Fit Decreasing

The pattern-based BFD can be implemented in the same way as the pattern-based FFD Algorithm 3. We just need to sort, in non-increasing order of gap, the list of terminal nodes before iterating through them in line 11. Theorem 1 ensures that the number of patterns in a BFD solution is limited by $2m$. The complexity of the previous algorithm is therefore $\mathcal{O}(m^2 \log m)$. For CSP instances, this is probably enough. However, in BPP instances, this algorithm runs in $\mathcal{O}(n^2 \log n)$. However, by storing the list of terminal nodes in a self-balanced binary search tree and using the upper bound operation, this algorithm can be implemented to run in $\mathcal{O}(\min(m^2, n) \log(\min(m, L)))$. Moreover, it is usually much faster than $\mathcal{O}(\min(m^2, n) \log(\min(m, L)))$ in cutting stock instances since the number of arcs is usually much smaller than $\mathcal{O}(\min(m^2, n))$ (see Section 3.5).

6 Computational results

All the algorithms were implemented in `Python` 2.6.1. The source code is available online (`http://www.dcc.fc.up.pt/~fdabrandao/code`). In the following tables, we present average results over 20 instances for each class u120, u250,...,t501 of OR-Library (2012)'s data set. These instances were introduced by Falkenauer (1996) for BPP and are divided in two classes. Uniform classes are composed by randomly generated instances, and triplets classes are harder problems in which the optimal solution is composed of rolls completely filled with exactly three pieces. Table 1 presents the meaning of each column in subsequent tables. In Table 2, we present the results using these instances with the original demand values. For these instances, the FFD solution can be obtained quickly using both assignment-based and pattern-based algorithms. In the Table 3, we present the results using the same piece lengths, but the demand for each length was multiplied by one million. In this case, the FFD/BFD solutions could only be obtained quickly (significatively less than one second in a standard desktop machine) using pattern-based algorithms, since the number of pieces is too large to allow the use of any conventional approach. Even a $\mathcal{O}(n \log n)$ algorithm is too slow since there are instances with one thousand million pieces. The exact solutions were obtained using the method described in Brandão (2012). In our experiments, FFD and BFD usually lead to solutions with approximately the same number of rolls. When this does not happen, BFD is usually better.

Table 1: Meaning of the data displayed in subsequent tables.

Label	Description
L	roll length
n	number of pieces
m	number of different piece lengths
R^*	optimum number of rolls
R^{FFD}, R^{BFD}	number of rolls in the FFD/BFD solution
$\%g^{\text{FFD}}$, $\%g^{\text{BFD}}$	relative gap $(R - R^*)/R^*$
$\#p^{\text{FFD}}$, $\#p^{\text{BFD}}$	number of different patterns in the FFD/BFD solution

Table 2: Bin packing results.

class	L	n	m	R^*	R^{FFD}	$\%g^{\text{FFD}}$	$\#p^{\text{FFD}}$	R^{BFD}	$\%g^{\text{BFD}}$	$\#p^{\text{BFD}}$
u120	150	120	63.20	49.05	49.75	1.4	40.95	49.75	1.4	40.90
u250	150	250	77.20	101.60	103.10	1.5	62.20	103.10	1.5	62.20
u500	150	500	80.80	201.20	203.90	1.3	80.60	203.90	1.3	80.70
u1000	150	1,000	81.00	400.55	405.40	1.2	95.25	405.40	1.2	95.25
t60	1,000	60	50.00	20.00	23.20	16.0	23.20	23.20	16.0	23.20
t120	1,000	120	86.20	40.00	45.80	14.5	45.80	45.80	14.5	45.80
t249	1,000	249	140.10	83.00	95.00	14.5	93.30	95.00	14.5	93.30
t501	1,000	501	194.20	167.00	190.05	13.8	171.10	190.05	13.8	171.10

Table 3: Cutting stock results.

class	n	R^*	R^{FFD}	$\%g^{\text{FFD}}$	$\#p^{\text{FFD}}$	R^{BFD}	$\%g^{\text{BFD}}$	$\#p^{\text{BFD}}$
u120	120,000,000	48,496,486.10	49,369,514.25	1.8	66.10	49,369,514.25	1.8	66.75
u250	250,000,000	101,089,303.40	102,636,835.85	1.5	88.55	102,636,835.85	1.5	88.85
u500	500,000,000	200,636,290.20	203,496,389.45	1.4	94.90	203,496,389.45	1.4	94.90
u1000	1,000,000,000	400,006,333.65	404,912,242.50	1.2	96.30	404,912,242.50	1.2	96.30
t60	60,000,000	20,000,000.00	22,783,333.75	13.9	65.95	22,783,333.75	13.9	65.95
t120	120,000,000	40,000,000.00	45,410,417.10	13.5	111.85	45,410,417.10	13.5	111.85
t249	249,000,000	83,000,000.00	94,579,166.95	14.0	176.35	94,579,166.95	14.0	176.35
t501	501,000,000	167,000,000.00	189,714,583.65	13.6	239.40	189,714,583.65	13.6	239.40

7 Conclusions

We presented pattern-based first and best fit decreasing (FFD/BFD) algorithms for cutting stock problems (CSP) that overcome the limitations of assignment-based algorithms. Conventional assignment-based algorithms are not polynomial in the CSP input size in its most common format (i.e., items of same size grouped into orders with a required level of demand). Therefore, with large demands, even for small CSP instances they are likely to be slow for computing FFD/BFD solutions. In the CSP, n pieces of m different lengths must be cut from rolls with length L. The pattern-based algorithms presented do not make assignments to rolls piece-by-piece and, therefore, they can be faster than linear in the number of items. The efficient pattern-based FFD/BFD implementations run in polynomial time in the cutting stock input size. These algorithms are as fast as the best known FFD/BFD implementations in BPP instances and they are much faster than any conventional assignment-based algorithm in CSP instances.

References

Bayer, R. (1972). Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Inf.*, 1:290–306.

Brandão, F. (2012). Bin Packing and Related Problems: Pattern-Based Approaches. Master's thesis, Faculdade de Ciências da Universidade do Porto, Portugal.

Coffman, Jr., E. G., Garey, M. R., and Johnson, D. S. (1997a). Approximation algorithms for bin packing: A survey. In Hochbaum, D. S., editor, *Approximation Algorithms for NP-hard Problems*, pages 46–93. PWS Publishing Co., Boston, MA, USA.

Coffman, Jr., E. G., Johnson, D. S., Mcgeoch, L. A., and Weber, R. R. (1997b). Bin Packing with Discrete Item Sizes Part II: Average-Case Behavior of FFD and BFD. *In preparation*, 13:384–402.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms*. The MIT Press, 2 edition.

Falkenauer, E. (1996). A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics*, 2:5–30. 10.1007/BF00226291.

Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

Johnson, D. S. (1973). *Near-optimal bin packing algorithms*. PhD thesis, MIT, Cambridge, MA.

OR-Library (2012). Retrieved January 1, 2013, `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`.

Wäscher, G., Haußner, H., and Schumann, H. (2007). An improved typology of cutting and packing problems. *European Journal of Operational Research*, 183(3):1109–1130.