A Thousand Ways to Pack the Bin - A Practical Approach to Two-Dimensional Rectangle Bin Packing

Jukka Jylänki

February 27, 2010

Abstract

We review several algorithms that can be used to solve the problem of packing rectangles into two-dimensional finite bins. Most of the presented algorithms have well been studied in literature, but some of the variants are less known and some are apparently regarded as "folklore" and no previous reference is known. Different variants are presented and compared. The main contribution of this survey is an original classification of these variants from the viewpoint of solving the finite bin packing problem. This work focuses on empirical studies on the problem variant where rectangles are placed orthogonally and may be rotated by 90 degrees. Synthetic tests are used as the main benchmark and solving a practical problem of generating texture atlases is used to test the real-world performance of each method. As a related contribution, an original proof concerning the number of maximal orthogonal rectangles inside a rectilinear polygon is presented.

Keywords: Two-dimensional bin packing, optimization, heuristic algorithm, on-line algorithm, NP-hard

Contents

1 Introduction

The two-dimensional rectangle bin packing is a classical problem in combinatorial optimization. In this problem, one is given a sequence of rectangles $(R_1, R_2, \ldots, R_n), R_i = (w_i, h_i)$ and the task is to find a packing of these items into a minimum number of bins of size (W, H). No two rectangles may intersect or be contained inside one another. This problem has several real-world applications and is proven to be NP-hard [1] by a reduction from the 2-partition problem [2]. There does not even exist an asymptotic polynomial time approximation scheme (APTAS), but it is APX-hard [3]. A lot of work has been done to develop efficient heuristic algorithms that approximate the optimal solution. In this survey we present several of these algorithms and compare their performance on a practical level. By changing only a small rule in the heuristic decisions of an algorithm one can obtain very different results in the produced packings. Most of the conducted research focuses on asymptotic performance ratios and typically neglects these subtleties, since they don't usually play a role in the theoretical properties of the algorithm. We welcome these kinds of changes and test in practice how they affect the quality of the produced packings.

The two-dimensional bin packing problem is a generalization of the one-dimensional bin packing problem, on which Csirik and Woeginger [4] give a good survey. For the two-dimensional problem, there exist several variants. In one version, the process is modelled as if the rectangles are received from some input one at a time, and they must immediately be placed into one of the bins without any knowledge of the upcoming items. This variant is called **online** rectangle bin packing. The opposite to this variant is the **offline** rectangle bin packing problem, in which the whole sequence to pack is known in advance. We examine algorithms for both variants.

In one formulation of the bin packing problem there may exist several simultaneously **open** bins, between which the algorithm can choose the destination for the current rectangle. In the more restricting variant, there is a limit on the number of bins that may be open at any given time, and to open a new one, an existing bin must be **closed**. The **-BNF** algorithms we will present can be used for the most restricting case where only one bin may be open at a time, but other variants exploit the case when there is no limit on the number of open bins.

A packing is called **orthogonal** if all the sides of the placed rectangles are parallel with the bin edges. We only consider packings that are **orthogonal** and we allow that each rectangle may be rotated by 90 degrees. This is called **rotatable** rectangle bin packing. That is, the packing algorithm may choose

for an input $R = (w, h)$ whether to pack the rectangle $R' = (h, w)$ instead. In some formulations of the bin packing problem, this is not allowed. This is not in any way a critical property for the working of any of the heuristic methods and each of them can be fit to work for the non-oriented rectangle bin packing case as well.

In some real-world applications it is required that the packings that are produced are **guillotineable**. A packing P is guillotineable if it can be split into two parts P_1, P_2 with a single straight horizontal or vertical cut that doesn't cross any of the rectangles in the packing, and where both P_1 and P_2 are either guillotineable as well or only consist of at most a single rectangle each. Not all of the algorithms presented in this survey produce guillotineable packings, but we make a mention of which do. Lodi, Martello and Vigo [5] provide an overview and comparison of variants with and without guillotineability or rotatability properties.

As a practical aspect, we confine ourselves to solving the problem with all integral values. That is, the dimensions of the bin and the rectangles as well as the coordinates on which the rectangles may be placed must all be integers.

2 The Algorithms

In this chapter we introduce each data structure and algorithm that was included in the review. These algorithms are classified in groups based on the underlying data structure that is used to represent the packing process and the free space left in the bin. We start with the easiest and then proceed to the more complicated ones.

2.1 The Shelf Algorithms

The **Shelf** algorithms (or level algorithms) are unarguably the simplest methods one can use to produce packings. We define a **shelf** to be a sub-rectangle of the bin with width W_b and height h_s. As the packing proceeds, the free area of the bin is organized into shelves, bottom-up, in which the rectangles are placed from left to right. The last shelf (the topmost shelf) is called the **open shelf**. Since the rectangles are placed bottom-up, the area above the open shelf is always unused. This allows that the height of the open shelf may be adjusted whenever a rectangle is placed on that shelf. For the shelves below the open shelf we don't have this freedom and those are called **closed** shelves.

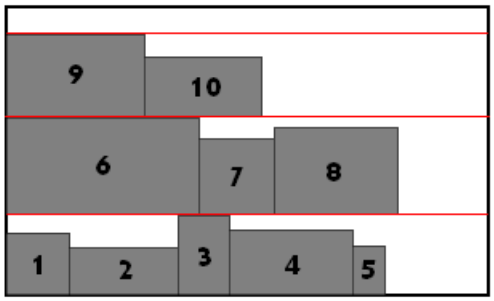

Figure 1: A sample packing produced by a Shelf algorithm.

When packing a rectangle (w, h) onto the shelf (W_b, h_s), we have to choose whether we rotate the rectangle or not, that is, whether the rectangle is stored in upright (store $(\min(w, h), \max(w, h))$) or sideways (store $(\max(w, h), \min(w, h))$) orientation. In all variants of our implementation, the choice is made in the following order:

1. If the rectangle is the first rectangle on a new open shelf, store it sideways. This is to minimize the height of the new shelf.
2. If the rectangle fits upright, i.e. if $\max(w, h) < h_s$, then store it so. This aims to minimize the wasted surface area between the rectangle top side and the shelf ceiling.
3. Otherwise store the rectangle sideways if possible.

The image 2.1 shows a sample packing produced by a shelf algorithm. The rectangles are numbered in the order they were placed in the bin and the red lines show the shelf ceilings. All the variants of the shelf algorithm generate packings very similar to the one shown in this image.

2.1.1 Shelf Next Fit (SHELF-NF)

Of all the algorithms and their variants presented in this paper, the Shelf Next Fit is absolutely the simplest method to produce a packing. It has a special property that no other algorithm reviewed here shares, namely that it only requires a constant amount of work memory. As it will be seen, all other algorithms use some kind of data structure that is at least linear in the number of rectangles already packed. For SHELF-NF, only

three temporary registers are needed. This property may be useful in some applications. Unfortunately, the packings produced by SHELF-NF can be quite far from the best methods presented in the paper.

Algorithm 1: SHELF-NF.

Initialize:
Set $y \leftarrow 0$.
Set $x \leftarrow 0$.
Set $h_s \leftarrow 0$.
Pack:
foreach *Rectangle* $R = (w, h)$ *in the sequence* **do**
    Determine the proper orientation.
    Try to fit the rectangle onto the current open shelf.
    If it does not fit, close the current shelf and open a new one.
    If there is no room for a new shelf, terminate.
end

Proposition 1. *The SHELF-NF algorithm can be implemented to run in* $\Theta(n)$ *time and* $O(1)$ *space.*

□

2.1.2 Shelf First Fit (SHELF-FF)

It is somewhat wasteful to forget about the free space in the old shelves when a new shelf is opened. Therefore all the variants of the SHELF-NF algorithm maintain a list of all the previously closed shelves so that rectangles can still be placed there if possible. But in case that there exists more than one shelf where the rectangle fits, which one should we pick? The policy in making this choice yields several variants. In Shelf First Fit we always place the rectangle into the shelf with the lowest index where it fits. This is quite straightforward, but note that now both the running time and memory consumption of the algorithm is linear in the number of shelves in the current bin.

With SHELF-FF, a rectangle that we manage to fit onto a closed shelf saves that space from being used in the open shelf. Compared to SHELF-NF, SHELF-FF can occasionally get a "free lunch" if it is able to pack a rectangle in this way. So one might think that SHELF-FF cannot perform worse than SHELF-NF, but this is not true. The reason is that since the packing decisions are heuristic in the first place, it cannot be guaranteed

that this smarter packing that SHELF-FF does would be any more optimal. In practice SHELF-FF performs better than SHELF-NF, but for some sequences it looks like SHELF-FF just hits a streak of bad luck when trying to outperform SHELF-NF and ends up with a worse packing. This effect is a recurring one when comparing other algorithms as well.

Proposition 2. *The SHELF-FF algorithm can be implemented to run in $O(n \log n)$ time and $O(n)$ space.*

Proof. The additional $O(n)$ space comes from having to store a data structure of the list of shelves, unlike in the SHELF-NF algorithm, where only the last shelf is kept track of. An implementation that finds the first shelf where the rectangle fits by linear search takes $O(n^2)$ time, but with a bisection method the shelf can also be found in $\log n$ time, thus giving a $O(n \log n)$ time algorithm. □

2.1.3 Shelf Best Width Fit (SHELF-BWF)

It can be seen as a shortcoming of SHELF-FF that it doesn't consider all the possible shelves as whole, but just greedily places the rectangle onto the first shelf it fits. Perhaps it is better to first look at all the possible shelves and only then pick a best one out of them. In Shelf Best Width Fit we take a rule of choosing the shelf in which the remaining width of the shelf space is minimized.

2.1.4 Shelf Best Height Fit (SHELF-BHF)

Since the edges dividing shelves are straight lines, packing a rectangle of smaller height than the shelf height just produces a strip of wasted space between the rectangle top side and the shelf ceiling. To minimize this wasted area, Shelf Best Height Fit chooses to pack each rectangle onto the shelf that minimizes the leftover height $h_s - h$.

2.1.5 Shelf Best Area Fit (SHELF-BAF)

Both of the above methods have their advantages. To try to combine them both we can try to maximize the total used shelf area. This results in the Shelf Best Area Fit algorithm.

2.1.6 Shelf Worst Width Fit (SHELF-WWF)

While SHELF-BWF tries to fill the width of each shelf as well as possible, the Shelf Worst Width Fit algorithm tries to do exactly the opposite and keep each shelf with as much width still available as possible. This is an another curiosity with heuristic algorithms. SHELF-WWF and SHELF-BWF are the total opposites of each other, but even still one cannot claim that one would be more optimal than the other.

With SHELF-WWF we adopt an extra rule that if we are packing a rectangle of width w and we find a shelf that has exactly w units of space still left, we immediately pick that shelf to pack the rectangle in.

Following the same pattern, one could define the algorithms Shelf Worst Height Fit and Shelf Worst Area Fit. But since the shelf algorithms waste the space between each packed rectangle and the shelf ceiling, trying to maximize this difference would correspond to maximizing wasted area, and therefore these variants are most likely suboptimal. If co-used with the Floor-Ceiling variant or with the Waste Map Improvement (see the next two subsections) this might not be strictly the case, but we did not test these variants nevertheless.

Proposition 3. *Each of the algorithms SHELF-BWF, SHELF-BHF, SHELF-BAF, SHELF-WWF can be implemented to run in $O(n^2)$ time and $O(n)$ space.*

Proof. For each rectangle to be packed, we examine each shelf to find the best of them. The number of shelves has a growth rate of $O(n)$. □

2.1.7 Shelf Floor-Ceiling

All the abovementioned variants still have the same problem that they cannot recover the free area that they waste when the rectangle heights do not match the height of the shelf. To fix this, Lodi, Martello and Vigo [6] proposed the Shelf Floor-Ceiling variant, where the input is sorted by decreasing long side first, and is packed normally into shelves proceeding left-to-right along the floor of the shelf. As soon as we close a shelf and thus fix the height of that shelf, we also start packing rectangles from right-to-left along the shelf ceiling. Since the input is sorted by decreasing height, tracking valid ceiling positions to place the rectangles into is feasible by using a simple data structure. The authors show that this improves the performance of the Shelf algorithm.

We did not implement the Shelf Floor-Ceiling, mostly because we feel that it is quite similar to and probably outperformed by the Skyline algorithm, but we cannot cannot verify this claim.

2.1.8 The Waste Map Improvement (-WM)

Another method to try to utilize the excessive wasting of free area in the Shelf algorithm is using what we call a Waste Map. Since the Guillotine algorithm presented in the next subsection is such a simple and effective way of storing free areas of the bin, we utilize it to keep track of all the areas that would otherwise go to waste.

For the Shelf algorithm, the process is as follows. We start the packing by initializing the Shelf algorithm as usual, and by initializing as a substructure an instance of the Guillotine packer algorithm. For a description of the Guillotine data structure and related algorithms, see the chapter 2.2 below. This data structure initially has $\mathcal{F} = \emptyset$. Whenever we close a shelf, we find all the disjoint rectangles of free area on that shelf and add those to $\mathcal{F}$. When packing a rectangle, we first check if the Guillotine packer can place the rectangle, and if not, we use the Shelf algorithm as usual. Then the question is that which variant of the Guillotine packer should we use? Since there are so many and to keep down the number of combinations to test, we only consider a few of the best performing ones.

Proposition 4. *The algorithms SHELF-x-WM can be implemented to run in $O(n^2)$ time and $O(n)$ space.*

Proof. For each rectangle, we first check if it can be packed into the GUILLOTINE data structure (see the next section). This can be done in linear time. If it doesn't fit, we do another linear search to find the appropriate shelf. An update of both the SHELF and the GUILLOTINE data structures is performed in constant time. Hence, the total running time is $O(n^2)$ time, requiring $O(n)$ space. □

Note that the time complexity of the SHELF-FF-WM algorithm is $O(n^2)$ and not $O(n \log n)$, since the Guillotine placement step dominates the binary search step when finding the destination shelf.

Figure 1 summarizes the algorithms presented in this chapter.

2.2 The Guillotine Algorithms

No matter what kind of tweaks are used to improve the Shelf method, it can still waste a lot of space in the worst case. In this chapter we pick a

Algorithm Name	Time Complexity	Space Complexity
SHELF-NF	$\Theta(n)$	$O(1)$
SHELF-FF	$O(n \log n)$	$O(n)$
SHELF-BWF	$O(n^2)$	$O(n)$
SHELF-BHF	$O(n^2)$	$O(n)$
SHELF-BAF	$O(n^2)$	$O(n)$
SHELF-WWF	$O(n^2)$	$O(n)$
SHELF-x-WM	$O(n^2)$	$O(n)$

Table 1: A summary of the different SHELF variants and their algorithmic complexities.

totally different approach to the problem. This algorithm is based on an operation that we call the **guillotine split placement**, which is a procedure of placing a rectangle to a corner of a free rectangle of the bin, after which the remaining L-shaped free space is split again into two disjoint free rectangles. This procedure and the possible split choices are shown in diagram 2.2. The actual process of packing several rectangles is then modelled as an iterative application of the guillotine split placement operation.

Algorithm based on this split rule are well known and widely used. For example, it is presented in the book 3D Games, Volume 2 [7] and also by several web authors such as Jim Scott [8] and John Ratcliff [9]. However, we could not find a source referring to the original author of this method or even less get a name for the algorithm. Therefore we name this method the **Guillotine** algorithm, since it produces packings that are easily seen to be guillotineable.

The Guillotine algorithm itself works as follows. We maintain a list of rectangles $\mathcal{F} = \{F_1, \ldots, F_n\}$ that represent the free space of the bin. These rectangles are pairwise disjoint, i.e. $F_i \cap F_j = \emptyset$ for all $i \neq j$ and the total free unused area of the bin can be computed with $\bigcup_{i=1}^{n} F_i$. The algorithm starts with a single free rectangle $\mathcal{F} = \{F_1 = (W, H)\}$. At each packing step, we first pick a free rectangle F_i to place the next rectangle $R = (w, h)$ into. The rectangle R is placed to the bottom-left corner of F_i, which is then split using the guillotine split rule to produce two smaller free rectangles F' and F'', which then replace F_i in the list of free rectangles. This procedure continues until no free rectangle can fit the next rectangle, and then the process is started again on a new empty bin. This algorithm is outlined in the diagram 2.

The Guillotine algorithm is very likeable since it keeps exact track of

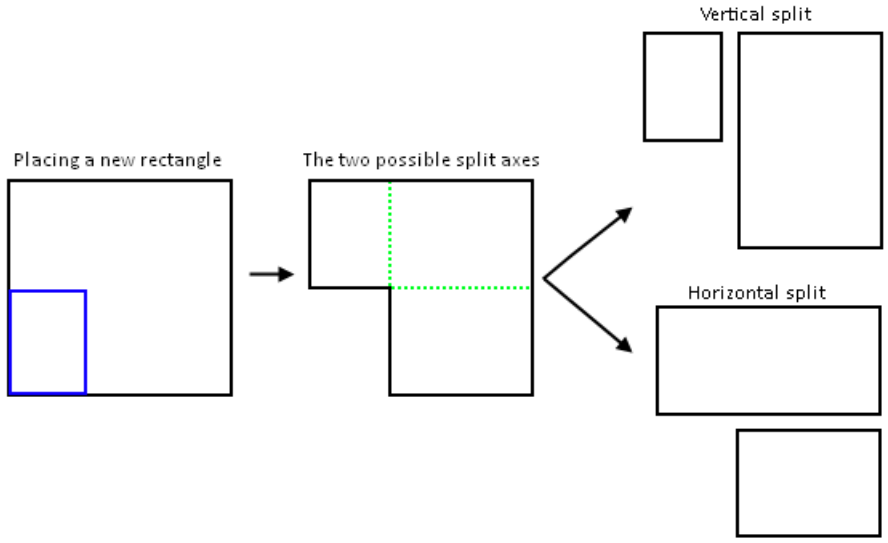

Figure 2: The guillotine split placement process. After placing a rectangle, there are two ways to store the remaining free area.

Algorithm 2: The Guillotine algorithm.

Initialize:
Set $\mathcal{F} = \{(W, H)\}$.
Pack:
foreach *Rectangle* $R = (w, h)$ *in the sequence* **do**
    Decide the free rectangle $F_i \in \mathcal{F}$ to pack the rectangle into.
    If no such rectangle is found, restart with a new bin.
    Decide the orientation for the rectangle and place it at the bottom-left of F_i.
    Use the guillotine split scheme to subdivide F_i into F' and F''.
    Set $\mathcal{F} \leftarrow \mathcal{F} \cup \{F', F''\} \setminus F_i$.
end

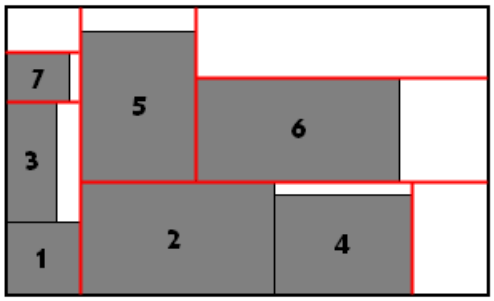

Figure 3: A sample packing produced by a Guillotine algorithm. The red lines denote the split choices.

the free areas of the bin and never "forgets" any free space, unlike the Shelf algorithms. The drawback here is that the algorithm only considers placements in which a rectangle R fully fits inside a single free rectangle F_i. It never tries to pack R into a position where it would straddle a split line. In other words, it fails to pack R if $R \not\subseteq F_i$ for all i, but $R \subseteq \bigcup_{i=1}^{n} F_i$.

A sample packing produced by the Guillotine algorithm is shown in image 2.2. The red lines denote the split lines that were used to cut the free area so that it can be represented using a set of disjoint rectangles. At this stage the set of free rectangles $\mathcal{F}$ consists of 8 rectangles, which correspond to the white areas of the image.

To complete the algorithm we still have to define two rules. First, we have to come up with a rule of how we select the F_i in which the rectangle is placed. Second, we have to choose which of the two possible directions we use for the split. We find six different ways to do both. These choices can be made independently, so this gives 36 different variants of the algorithm. It is not obvious whether one convention would be superior to another, so we test them all.

When reviewing published implementations of this algorithm, we found that some of them [8] [7] construct elaborate data structures that utilise kD-trees, binary partitioning or recursion to make the choice of selecting the free rectangle. Our previous published implementation [10] operated in a similar way as well. We feel this is overly complicated, unnecessary and outright suboptimal, since these data structures do not generally allow one to well-define an efficient rule for selecting the next free rectangle. In the implementation written for this review, we have switched to using a resizable array to store the free rectangles. For optimization purposes the

array could be stored in sorted order to allow a loop early-out optimization, but after observing good enough practical performance we did not bother with such details.

In the following subchapters, we present the different heuristic selection rules we used for the review.

2.2.1 Guillotine Best Area Fit (GUILLOTINE-BAF)

Very similarly to SHELF-BAF, the **Guillotine Best Area Fit** picks the free rectangle F_i of smallest area in which the next rectangle fits. This is a natural rule to try to minimize the narrow strips of wasted space.

2.2.2 Guillotine Best Short Side Fit (GUILLOTINE-BSSF)

When we are placing a rectangle $R = (w, h)$ into a free rectangle $F_i = (w_f, h_f)$, we can consider the differences in the side lengths of these two rectangles. The **Guillotine Best Short Side Fit** rule chooses to pack R into such F_i that $\min(w_f - w,\ h_f - h)$ is the smallest. In other words, we minimize the length of the shorter leftover side.

2.2.3 Guillotine Best Long Side Fit (GUILLOTINE-BLSF)

We get another rule with **Guillotine Best Long Side Fit**, where pack R into an F_i such that $\max(w_f - w,\ h_f - h)$ is the smallest. That is, we minimize the length of the longer leftover side.

2.2.4 Guillotine Worst Fit Rules

Since the Worst Fit variants for the Shelf algorithm were not a total bust, we can try the same approach here. These Worst Fit rules are analogous to the Best Fit rules in the previous subsection. The **Guillotine Worst Area Fit** (GUILLOTINE-WAF) algorithm packs R into the F_i such that the area left over is maximized. Note that with this variant, as well as with all other Guillotine variants, we have the special placement rule that if $R = F_i$ for some i, then that F_i is picked immediately, since it is the perfect match.

The Worst Width Fit variant for the Shelf algorithm can be brought over to the Guillotine algorithm in two different ways. In **Guillotine Worst Short Side Fit** (GUILLOTINE-WSSF), we maximize the length of the shorter leftover side. Finally, the third possible variant is the **Guillotine Worst Long Side Fit** (GUILLOTINE-WLSF), in which we maximize the length of the longer leftover side.

The essential motivation for all Worst Fit variants is the same as with the Shelf Worst Fit variants - to try to keep big spaces left in the free rectangles as long as possible and to try to avoid very small useless strips of space.

Proposition 5. *The algorithms GUILLOTINE-BAF, -BSSF, -BLSF, -WAF, -WSSF and -WLSF can be implemented to run in $O(n^2)$ time and $O(n)$ space.*

Proof. These algorithms only differ by how they compare two elements of $\mathcal{F}$, which is in each case a constant time operation. The size of the free rectangle structure $|\mathcal{F}|$ has a growth rate of $O(n)$, since at each packing step we add at most one new free rectangle into $\mathcal{F}$. For each rectangle, we examine each of the free rectangles in $\mathcal{F}$ one at a time, which yields the running time $O(n^2)$. □

2.2.5 The Rectangle Merge Improvement (-RM)

The biggest issue with the Guillotine algorithm is that rectangles cannot be placed in any position of the free area where the rectangle would straddle an existing split line. If the free space is sufficiently fragmented, the algorithm can incorrectly report that there is no free space to place a rectangle even though there is. Therefore we assume we would get better packings if we could minimize the number of split lines dividing the free area. However, it is not obvious if much can be done to mend this since we insist to represent the free area using a set of disjoint rectangles. There is a straightforward procedure that we simply call the **Rectangle Merge Improvement**. The way it works is that after packing a rectangle, we go through all the free rectangles and see if there exists a pair of neighboring rectangles F_i, F_j such that $F_i \cup F_j$ can be exactly represented by a single bigger rectangle. If so, we merge these two into one, which effectively removes fragmentation of the free area by removing a single split line that existed between F_i and F_j. In his online blog John Ratcliff writes [9] to imply that this process is important for robustness, so we test all the variants with and without this improvement.

Proposition 6. *The algorithms GUILLOTINE-x-RM can be implemented to run in $O(n^3)$ time and $O(n)$ space.*

Proof. After packing each rectangle, we do a rectangle merge step by examining each pair $F_i, F_j \in \mathcal{F}$. There are $\Theta(n^2)$ such pairs and this step rises to dominate the overall complexity. □

2.3 Split Rules for the Guillotine Algorithm

Since the split axes determine the sizes of the free rectangles and because a placement of a rectangle may not straddle a split line, it is important to be careful about how the splits are performed. In this subsection we present different methods of choosing whether to split horizontally or vertically. In the following, let $F_i = (w_f, h_f)$ be the free rectangle inside which the rectangle $R = (w, h)$ has just been packed.

As all of the following split rules only make a local constant time choice of the direction of the split, they don't affect the complexity of the main algorithm.

2.3.1 Shorter/Longer Axis Split Rule (-SAS, -LAS)

As the simplest convention, we can determine the split axis independent of the dimension of R and just split horizontally if $w_f < h_f$ and vertically otherwise. This is called the Shorter Axis Split Rule (-SAS). As the opposite rule, the Longer Axis Split Rule (-LAS) splits horizontally if $w_f \geq h_f$ and vertically otherwise.

2.3.2 Shorter/Longer Leftover Axis Split Rule (-SLAS, -LLAS)

We can also examine the leftover lengths $w_f - w$ and $h_f - h$ of the free rectangle. In the Shorter Leftover Axis Split Rule (-SLAS), we split horizontally if $w_f - w < h_f - h$, and vertically otherwise. Again, we can also take the opposite convention and in the Longer Leftover Axis Split rule (-LLAS), we split horizontally if $w_f - w \geq h_f - h$, and vertically otherwise.

2.3.3 Max/Min Area Split Rule (-MAXAS, -MINAS)

Instead of looking at the side lengths, we can also examine the surface areas of the four subrectangles that are formed in the process. Diagram Y shows this setting. In the Max Area Split Rule (-MAXAS), we try to keep the rectangles A_1 and A_2 as even-sized as possible and join A_3 with the smaller of these two. With the Min Area Split Rule (-MINAS) we join A_3 with the larger of A_1 and A_2 to produce a single larger free rectangle instead.

We refer to each Guillotine variant using a name of the form GUILLOTINE-*RECT-SPLIT*, where RECT is one of the strings BAF, BSSF, BLSF, WAF, WSSF or WLSF, and SPLIT is one of the strings SAS, LAS, SLAS, LLAS, MAXAS, MINAS. If the Rectangle Merge improvement is used, we append

Algorithm Name	Time Complexity	Space Complexity
GUILLOTINE-*RECT-SPLIT*	(n^2)	$O(n)$
GUILLOTINE-*RECT-SPLIT*-RM	(n^3)	$O(n)$

Table 2: A summary of the different GUILLOTINE variants and their algorithmic complexities.

the suffix -RM to the name. To finish this chapter, table 2 shows a summary of these algorithms.

2.4 The Maximal Rectangles Algorithms

The Guillotine algorithm introduced in the previous section is a big improvement over the Shelf algorithm, but the split line boundaries still cause problems with the practical performance. To try to remove all these issues altogether, we introduce the **Maximal Rectangles** algorithm. This algorithm is in some sense based on an extension of the Guillotine Split Placement rule. Like the Guillotine algorithm, the Maximal Rectangles algorithm stores a list of free rectangles that represents the free area of the bin. But unlike the Guillotine algorithm which chooses one of the two split axes, the Maximal Rectangles algorithm performs an operation that essentially corresponds to picking both split axes at the same time.

This split process is shown in the diagram 2.4. When we place an input rectangle R to the bottom-left of a free rectangle F, we compute the two rectangles F_1 and F_2 that cover the L-shaped region of $F \setminus R$ and update $\mathcal{F} \leftarrow (\mathcal{F} \cup \{F_1, F_2\}) \setminus \{F\}$. The 'Maximal' in the name of the algorithm refers to the property that these new rectangles F_1 and F_2 are formed to be of maximal length in each direction. That is, at each side they touch either the bin edge or some rectangle already placed into the bin. Performing the split in this way gives us the following special property for the list $\mathcal{F}$.

Proposition 7. *Let $\mathcal{F} = \{F_1, \ldots, \ldots, F_n\}$ be the set of maximal free rectangles that represents the free area left in the bin at some packing step of the Maximal Rectangles algorithm. Then for any rectangle $R \subseteq \bigcup_{i=1}^{n} F_i$, there exists $F_i \in \mathcal{F}$ such that $R \subseteq F_i$.*

□

The above proposition guarantees that when considering the potential positions to place a rectangle to, we can just consider each free rectangle F_i

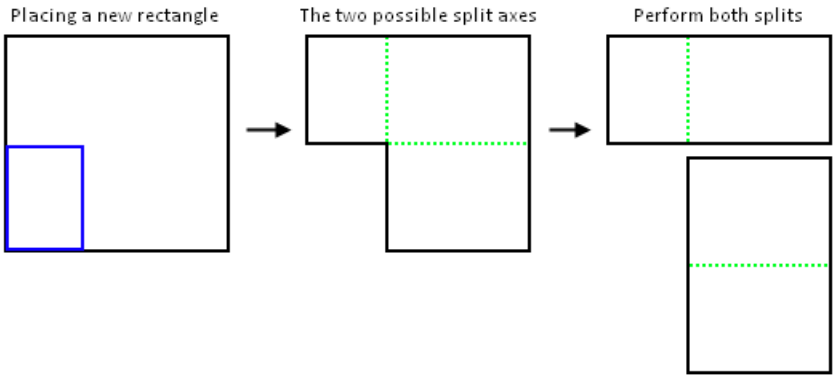

Figure 4: The rectangle placement rule for the MAXRECTS data structure. Both the rectangles on the right are stored in $\mathcal{F}$.

in turn and be sure that if the rectangle fits the bin we will not miss a valid placement.

Losing the property that the free rectangles F_i are pairwise disjoint generates issues when placing a rectangle. This is because after we have packed R into some F_i, we have to check and update all the other rectangles $F_j \in \mathcal{F}$ for which $R \cap F_j \neq \emptyset$, or our data structure becomes inconsistent. We do this simply by looping through each free rectangle F_j and intersecting it with R, producing a set of new free rectangles. After this step we may be left with degenerate and/or nonmaximal rectangles in the set $\mathcal{F}$, so we go through each free rectangle $F_i \in \mathcal{F}$ again and remove it if there exists another rectangle $F_j \in \mathcal{F}, i \neq j$, for which $F_i \subseteq F_j$.

Algorithm 3: The Maximal Rectangles algorithm.

```
Initialize:
Set F = {(W, H)}.
Pack:
foreach Rectangle R = (w, h) in the sequence do
    Decide the free rectangle F_i ∈ F to pack the rectangle R into.
    If no such rectangle is found, restart with a new bin.
    Decide the orientation for the rectangle and place it at the
    bottom-left of F_i. Denote by B the bounding box of R in the bin
    after it has been positioned.
    Use the MAXRECTS split scheme to subdivide F_i into F' and F''.
    Set F ← F ∪ {F', F''} \ {F_i}.
    foreach Free Rectangle F ∈ F do
        Compute F \ B and subdivide the result into at most four
        new rectangles G_1, ..., G_4.
        Set F ← F ∪ {G_1, ..., G_4} \ {F}.
    end
    foreach Ordered pair of free rectangles F_i, F_j ∈ F do
        if F_i contains F_j then
            Set F ← F \ {F_j}
        end
    end
end
```

2.4.1 Maximal Rectangles Bottom-Left (MAXRECTS-BL)

A very different variant to the algorithms defined in the previous sections is what is called the Bottom-Left algorithm, or the Tetris algorithm. The heuristic rule used by this algorithm is very simple: Orient and place each rectangle to the position where the y-coordinate of the top side of the rectangle is the smallest and if there are several such valid positions, pick the one that has the smallest x-coordinate value. We can use the Maximal Rectangles data structure to implement this algorithm and it will be called the **Maximal Rectangles Bottom-Left** algorithm. See Bernard Chazelle's paper [11] on a more efficient implementation of this algorithm.

Image 2.4.1 shows a sample output produced by the MAXRECTS-BL algorithm. The maximal rectangles inside the free area are colored in red, green and blue, and slightly shrunk for clarity.

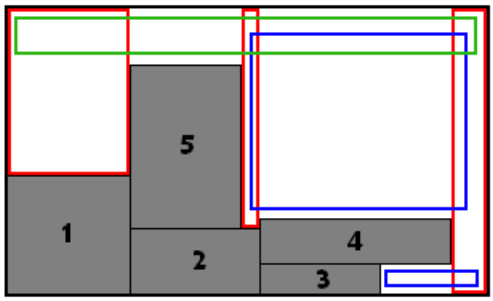

Figure 5: A sample packing produced by the MAXRECTS-BL algorithm. The maximal rectangles of $\mathcal{F}$ are shown in colors.

2.4.2 Maximal Rectangles Best Area Fit (MAXRECTS-BAF)

We can use the same heuristic rules when choosing the free rectangle in the Maximal Rectangles data structure as we had with the Guillotine algorithm. In **Maximal Rectangles Best Area Fit** we pick the $F_i \in \mathcal{F}$ that is smallest in area to place the next rectangle R into. If there is a tie, we use the Best Short Side Fit rule to break it.

2.4.3 Maximal Rectangles Best Short Side Fit (MAXRECTS-BSSF)

Again, we can also consider the differences in the side lengths of R and F_i. As was with the GUILLOTINE-BSSF, the **Maximal Rectangles Best Short Side Fit** rule chooses to pack R into such F_i that $\min(w_f - w, h_f - h)$ is the smallest. In other words, we minimize the length of the shorter leftover side.

2.4.4 Maximal Rectangles Best Long Side Fit (MAXRECTS-BLSF)

The **Maximal Rectangles Best Long Side Fit** rule is exactly analogous. We pack R into an F_i such that $\max(w_f - w, h_f - h)$ is the smallest. That is, we minimize the length of the longer leftover side.

2.4.5 The Efficiency of MAXRECTS

Analysing the efficiency of algorithms that are based on the MAXRECTS data structure is not as straightforward, as is seen in this section.

Proposition 8. *The algorithms MAXRECTS-BL, -BAF, -BSSF, -BLSF can be implemented to run in $O(|\mathcal{F}|^2 n)$ time. They consume $\Theta(|\mathcal{F}|)$ space.*

Proof. After packing each rectangle and having intersected it with the elements of $\mathcal{F}$ and produced the set of new potential maximal rectangles, we go through each pair of elements in $\mathcal{F}$ to prune the redundant free rectangles from the list. This is the most time consuming step of the algorithm, yielding the $O(|\mathcal{F}|^2 n)$ time complexity. □

Based on the above, it is very important to know the growth rate of $|\mathcal{F}|$ in order to estimate the actual complexity of these algorithms. We do not know of any previous results on this problem, but are still able to settle the question. To our best knowledge, the result presented below is original, except for the proof on the lower bound of $|\mathcal{F}|$, which is a straightforward adaptation from a proof on a similar problem published by [12]. We start with a few preliminaries.

Definition A *rectilinear polygon* is a two-dimensional connected, closed and non-self-intersecting polygon consisting only of horizontal and vertical lines.

It is obvious that at each packing step, the free space of the bin forms one or more rectilinear polygons. The number of vertices in these polygons is linear in n, the number of rectangles we have packed. It is also immediate that the worst case occurs when the free space is not disconnected, but forms only a single rectilinear polygon, call it P. Denote by $p_1, \ldots, p_k$ the vertices of this polygon.

Let M be a maximal rectangle of P, and m be a side of M. We say that m is edge-supported if it does not touch any vertex of P. Otherwise, one or more vertices of P touch m and we say that m is vertex-supported. We can make the following observation.

Proposition 9. *For each maximal rectangle M, there exist two opposing sides of M that are both vertex-supported.*

Proof. If all the sides of M are vertex-supported, then the statement naturally holds. So assume that a side m of M is edge-supported instead. If a side adjacent to m were also edge-supported, then P would have to be self-intersecting, which is not allowed. Hence, the two sides that are adjacent to m must both be vertex-supported and again there exist two opposing sides of M that are vertex-supported. □

The above lemma gives us a constructive method for defining any of the maximal rectangles of P in terms of its two supporting vertices and the knowledge of whether these support the horizontal or vertical edges of the maximal rectangle.

Proposition 10. *A triplet (p_i, p_j, o), where p_i and p_j are vertices of P, and $o \in \{\boldsymbol{H}, \boldsymbol{V}\}$ is a binary label denoting the choice of expansion direction (horizontal or vertical), uniquely constructs a maximal rectangle of P.*

Proof. Fix $p_i = (x_i, y_i)$ and $p_j = (x_j, y_j)$ to be two vertices of P. Let R be the rectangle with the bottom-left coordinate $(\min(x_i, x_j), \min(y_i, y_j))$ and the top-right coordinate $(\max(x_i, x_j), \max(y_i, y_j))$. R cannot intersect P, or p_i and p_j are not "compatible", and do not form a maximal rectangle. Then, if $o = \mathbf{H}$, the height of the maximal rectangle is $|y_i - y_j|$ and there is only one way to expand the left and right sides of R to form a maximal rectangle. Equivalently, if $o = \mathbf{V}$, then the width of the maximal rectangle is $|x_i - x_j|$ and there is again only one way to expand the top and bottom sides of R to form a maximal rectangle. □

Since each maximal rectangle is characterized by a triplet of the above form, we can give an upper bound on the number of different maximal rectangles that can exist. Hence, we have obtained the following result.

Corollary 2.1. *The number of maximal rectangles in a rectilinear polygon with n vertices is at most $2n^2$.*

□.

Having an upper bound for the number of maximal rectangles is not that useful if the bound is loose. The next result shows that in fact this bound is asymptotically tight. The proof is a straightforward adaptation from the example presented in [12].

Proposition 11. *The upper bound given in 2.1 is asymptotically tight in the worst case.*

Proof. We prove this by giving an instance of a rectilinear polygon with n vertices where the number of maximal rectangles is proportional to n^2.

This instance is shown in diagram 2.4.5. In this polygon, there exists two "staircases", which both have a number of corners linear to n. These staircases have been specially positioned (see the dotted helper lines) so that every corner in the upper-left part of the polygon forms a pair with

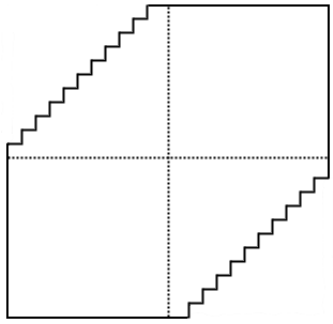

Figure 6: An example of a worst case configuration producing $O(n^2)$ maximal rectangles.

all the corners on the bottom-left, giving a number of maximal rectangles quadratic to n.

□

Combining the results 8 and 2.1 above show a time complexity of $O(n^5)$ for the MAXRECTS algorithm. Proposition 11 shows that this bound is tight if we consider arbitrary rectilinear polygons, but in practice the polygons formed in the packing process behave much more nicely. In our tests we have observed that the size of $\mathcal{F}$ is linear in n, which would suggest an average $O(n^3)$ time and $O(n)$ space complexity for the whole algorithm. Still, the MAXRECTS-BL variant is beaten by Chazelle's excellent $O(n^2)$ time and $O(n)$ space implementation [11], which is based on a representation of the free space by using trees of doubly linked-lists.

2.4.6 Maximal Rectangles Contact Point (MAXRECTS-CP)

Lodi, Martello and Vigo [5] describe an interesting variant that is unique to the ones presented already. In **Maximal Rectangles Contact Point** we look to place R into a position where the length of the perimeter of R that is touched by the bin edge or by a previously packed rectangle is maximized. We only considered bottom-left stable rectangle placements in this algorithm. In [5], this algorithm is called the **Touching Perimeter** algorithm.

Proposition 12. *The algorithm MAXRECTS-CP can be implemented to run in $O(|\mathcal{F}|^2 n)$ time and $\Theta(|\mathcal{F}|)$ space as well.*

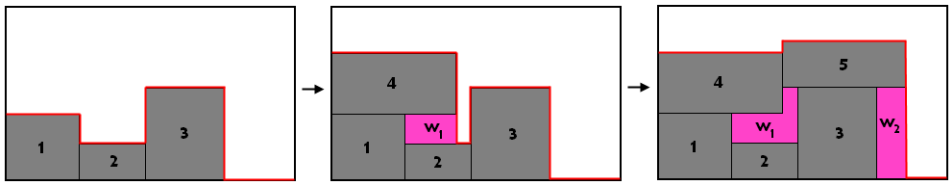

Figure 7: A sample packing produced by the SKYLINE-BL algorithm.

Proof. The only issue compared to the other MAXRECTS-x algorithms is in the scoring procedure of MAXRECTS-CP, which involves going through the list of all previously packed rectangles. This is a linear step that is performed for each rectangle that is to be packed. However, since pruning $\mathcal{F}$ is of time complexity $\Theta(|\mathcal{F}|^2)$, the time taken by this linear step is just negligible in comparison. The space consumption of this algorithm is exactly the same as with MAXRECTS-x. □

2.5 The Skyline Algorithms

Since the Maximal Rectangles algorithm involves some tedious manipulation to maintain the list of maximal free rectangles, we propose a simplified data structure that can also be used to implement the Bottom-Left heuristics. The data structure in the **Skyline** algorithm is "lossy", just like with the Shelf algorithms, that is, it cannot perfectly keep track of the free areas of the bin and may mark some unused space as used. As a trade-off, the Skyline algorithms produce packings a lot faster than the ones using the Maximal Rectangles data structure.

The way the Skyline data structure works is that it only maintains a list of the horizon or "skyline" edges formed by the topmost edges of already packed rectangles. This list is very simple to manage and grows linearly in the number of the rectangles already packed.

Wei et al. [13] describe a very similar method. In their approach they call the skyline an *envelope*. The data structure is essentially the same, but the rule according to which they choose the placement position differs.

2.5.1 Skyline Bottom-Left (SKYLINE-BL)

The Skyline data structure allows us to implement the same Bottom-left heuristics that the MAXRECTS-BL does, except a bit of packing efficiency is traded for runtime performance. In **Skyline Bottom-Left**, we pack the

rectangle R left-aligned on top of that skyline level that results in the top side of R lie at the bottommost position. Since R can be rotated, this might not be the skyline level that lies in the lowest position.

2.5.2 Skyline Best Fit (SKYLINE-BF)

Since the Skyline data structure is prone to losing information about free areas, we can force as a heuristic to try to minimize this from happening as much as possible. This yields the **Skyline Best Fit** variant. In this variant, for each candidate position to pack the next rectangle into, we compute the total area of the bin that would be lost if the rectangle was positioned there. Then we choose the position that minimizes this loss. If there is a tie, we use the Bottom-Left rule to decide.

2.5.3 The Waste Map Improvement (-WM)

Since it is quite easy to compute the free rectangles that we will lose when packing a rectangle on top of a hole, we can utilize the Guillotine data structure to store this space and use it as a secondary data structure. This is exactly the same idea that was used with the Shelf algorithm.

Proposition 13. *The algorithms SKYLINE-x and SKYLINE-x-WM can be implemented to run in $O(n^2)$ time and $O(n)$ space.*

3 General Improvement Methods

In this section we consider methods that improve the packing performance independent of the actual algorithm we are using to produce the packing. What all the aforementioned algorithms have in common is that they work with online input. They place all the rectangles in the order they are given and never move a rectangle that has already been placed. There is no backtracking or any kind of search involved in making the choices. These kind of restrictions greatly simplify the complexity of the algorithms and also the effort required to implement them. The downside of course is that the quality of the packings can be rather poor in the worst case. In this section we consider a few tricks that can be used to greatly improve the situation.

3.1 Choosing the Destination Bin

We have not yet discussed how the algorithms work when the rectangles do not fit into a single bin and multiple bins must be used. These rules are very similar to the heuristics we use when picking the destination shelf in the Shelf algorithms. In **Bin Next Fit** (-BNF), we only have a single open bin into which rectangles are packed. When the next input rectangle does not fit into that bin, that bin is closed and completely forgotten about, and a new bin is opened. Effectively all the algorithms we have previously discussed are of type -BNF, since they have just dealt with one bin.

In the **Bin First Fit** (-BFF), we consider the bins in the order they were opened and pack the input rectangle into the bin with the lowest index where it fits. When using the **Bin Best Fit** (-BBF) rule, the rectangle is packed into the bin that gives the best score for whatever criterion the algorithm uses to decide between possible placements. Analogously to the shelf selection case, one can define **Bin Worst Fit** as well, but in this survey this variant was not implemented, since it was assumed to be suboptimal.

Proposition 14. *The -BFF variant adds a factor of* $\log n$ *to the complexity of the corresponding -BNF algorithm. For example, the running time of SHELF-FF-BNF is* $O(n \log n)$*, but the running time of SHELF-FF-BFF is* $O(n \log^2 n)$*.*

Proof. Since the number of rectangles each bin can hold is independent of n, the number of bins needed to pack n items is of $\Theta(n)$. Just like with SHELF-FF versus SHELF-NF, we can find the destination bin for the rectangle in $\log n$ time by using a bisection method (binary search). □

In the above proof, by substituting the binary search step with a linear search, we also get the following result.

Proposition 15. *The -BBF variant adds a factor of* n *to the complexity of the corresponding -BNF algorithm. For example, the running time of SHELF-FF-BNF is* $O(n \log n)$*, but the running time of SHELF-FF-BBF is* $O(n^2 \log n)$*.*

□

3.2 Sorting the Input

An easy method to enhance the performance of any online packing algorithm is to simply sort the sequence by some criterion before producing the

packing. Since this is just a preprocess step, it does not require any changes be made to the existing packing routine, which makes it very practical. Of course, this can be considered only if we know the whole sequence in advance.

We can think of several different methods to use as the comparison function for the sorting routine. If we have two rectangles $R_a = (w_a, h_a)$ and $R_b = (w_b, h_b)$, where $w_a \leq h_a$ and $w_b \leq h_b$, we can compare them at least in the following ways:

1. Sort by area. $R_a \prec R_b$ if $w_a h_a < w_b h_b$. This variant will be called -ASCA. We can of course reverse the condition to get the variant -DESCA.

2. Sort by the shorter side first, followed by a comparison of the longer side. $R_a \prec R_b$ if $w_a < w_b$ or if $w_a = w_b$ and $h_a < h_b$. These variants will be called -ASCSS and -DESCSS.

3. Sort by the longer side first, followed by a comparison of the shorter side. $R_a \prec R_b$ if $h_a < h_b$ or if $h_a = h_b$ and $w_a < w_b$. These variants will be called -ASCLS and -DESCLS.

4. Sort by perimeter. $R_a \prec R_b$ if $w_a + h_a < w_b + h_b$. These variants will be called -ASCPERIM and -DESCPERIM.

5. Sort by the difference in side lengths. $R_a \prec R_b$ if $|w_a - h_a| < |w_b - h_b|$. These variants will be called -ASCDIFF and -DESCDIFF.

6. Sort by the ratio of the side lengths. $R_a \prec R_b$ if $\frac{w_a}{h_a} < \frac{w_b}{h_b}$. These variants will be called -ASCRATIO and -DESCRATIO.

The sorting step takes $O(n \log n)$ time. Except for the algorithm SHELF-NF-BNF, the time to sort is dominated by the time taken to produce the actual packing and so the overall running time is not affected.

3.3 The Globally Best Choice

Most of the variants we have considered have a structure that can be represented as follows. Given the next input rectangle R' in the sequence of input rectangles $\mathcal{R}$, we have a set of choices $\mathcal{S}$ to place R' into and a scoring function $C : \mathcal{S} \times \mathcal{R} \mapsto \mathbb{R}$ defined by the heuristic rules of the variant. Then find $S' = \max_{S \in \mathcal{S}} C(S, R')$ and pack R according to the choice S'. The set $\mathcal{S}$ can be interpreted as follows:

1. For the Shelf algorithms, $\mathcal{S}$ is the set of shelves where R' can be validly placed onto.

2. For the Guillotine algorithms, $\mathcal{S}$ is the set of free rectangles $\mathcal{F}$ times the two choices for orienting R' that are valid placements. We pondered on even extending $\mathcal{S}$ to cover the two possible choices for choosing the split axis, but this was deemed to slow down the search too much.

3. For the Maximal Rectangles algorithms, $\mathcal{S}$ is the set of the maximal rectangles times the two choices for orienting R' that are valid placements.

4. For the Skyline algorithms, $\mathcal{S}$ is the set of skyline levels times the two choices for orienting R' that are valid placements.

There is a natural extension of this optimization rule that can lead to better packings. Instead of searching only the set $\{(S, R)|S \in \mathcal{S}\}$ where R is fixed, consider all the elements of $\mathcal{R}$, that is, search the whole set $\{(S, R)|S \in \mathcal{S}, R \in \mathcal{R}\}$. Then, at each packing step, we find $(S'', R'') = \max_{S \in \mathcal{S}, R \in \mathcal{R}} C(S, R)$ and pack R'' according to the choice S''.

Put in words, at each packing step, we go through each unpacked rectangle and each possible placement of that rectangle and compute the score for that particular placement. Then we pick the one that maximizes the value of the heuristic rule we are considering. Since instead of the next rectangle in the sequence we are picking the globally best one of all the rectangles still left, we give this variant the suffix -GLOBAL. Of course with this rule it does not matter any more in which order the sequence of input rectangles is given (ignoring any ties in the scoring function if there is no tiebreaker), so sorting is not performed with this variant.

Proposition 16. *The -GLOBAL variant adds a linear factor to the complexity of the corresponding online version of the algorithm. For example, the algorithm SKYLINE-BL can be implemented to run in $O(n^2)$ time, so the variant SKYLINE-BL-GLOBAL can be implemented to run in $O(n^3)$ time. Space complexity is not changed.*

Proof. We can think of this process like follows. At each packing step we pack all the rectangles still left in the input sequence in parallel, and then fix our choice to packing the one rectangle that gave the best value for the scoring function. The other parallel paths are then forgotten. The number of these parallel paths is of course linear in n, and thus the work done to pack a single rectangle is multiplied by n, giving the extra linear factor to the overall complexity. □

Applying these improvements to all of the algorithm variants from the previous chapters, we obtain the final list of algorithms for the review. Table 6 summarizes all of these.

4 Synthetic Benchmarks

We implemented most of the possible combinations of the rule variants presented in the previous sections, except for the algorithms that we explicitly mentioned to have been omitted. The number of variants we tested sums up to a grand total of 2619 distinct algorithms, which is too huge a number to present in a single tabular form. Therefore we present the results by first hand-picking different variants from each class and then present the set of "best contenders" in a final review.

Since the heart of all these algorithms consists of a heuristic rule, it is possible that one algorithm is better than the other for some input, but with another input sequence, the result is the opposite. Based on our tests this phenomenon is even more common than one might think, which makes it near impossible to single out one best algorithm. Still, some of the variants that we have described are clearly suboptimal and it is unlikely that there exist input sequences on which these variants could show their "full potential" and outperform the other algorithms. Moreover, some algorithms have been observed to consistently produce relatively good packings no matter what the input, while others seem to be more sensitive to the particular input sequence.

Because of this instability, to try to correctly estimate the relative performance of the different algorithms, we construct the input sequences using different selections of uniform probability distributions. The following section presents the method.

4.1 Rectangle Categories and Probability Distributions

To start with, we divide the possible side lengths of a rectangle into distinct categories. These categories are shown in table 3. To generate a side length from a given category, we use random uniform sampling.

Then we assemble categories of possible rectangle sizes using these side length categories. The rectangle categories we used are presented in table 4. Since we are interested in the problem variant where rotating the rectangles is allowed, the list of rectangle categories does not contain the cases where $h > w$.

Category Name	Length distribution
Tiny	$[1, \frac{1}{4}B]$
Short	$[\frac{1}{4}B, \frac{2}{4}B]$
Medium	$[\frac{2}{4}B, \frac{3}{4}B]$
Long	$[\frac{3}{4}B, B]$

Table 3: Categories for rectangle side lengths relative to the bin side B.

	Tiny	Short	Medium	Long
Tiny	R_1	R_2	R_3	R_4
Short		R_5	R_6	R_7
Medium			R_8	R_9
Long				R_{10}

Table 4: Categories for rectangle sizes (w, h).

Distribution	A	B	C	Distribution	A	B	C
D_1	90%	10%	-	D_{10}	40%	30%	30%
D_2	70%	30%	-	D_{11}	60%	-	40%
D_3	50%	50%	-	D_{12}	40%	20%	40%
D_4	80%	10%	10%	D_{13}	40%	10%	50%
D_5	60%	30%	10%	D_{14}	40%	-	60%
D_6	80%	-	20%	D_{15}	20%	20%	60%
D_7	60%	20%	20%	D_{16}	20%	10%	70%
D_8	40%	40%	20%	D_{17}	20%	-	80%
D_9	60%	10%	30%	D_{18}	-	-	100%

Table 5: Distributions for choosing rectangles from categories A, B and C.

To generate actual input sequences, we need to define the probability distributions according to which we select rectangles from each of the classes $R_1, \ldots, R_{10}$. Since the number of ways this can be done is enormous even if the probabilities are quantized, we pick the following scheme. We select two rectangle categories, A and B, and let $C = \{R_1, \ldots, R_{10}\} \setminus \{A, B\}$. To generate an actual problem instance, we draw rectangles from A, B and C according to the uniform distribution presented in table 5. As a special note, if the probability in the column B is marked with "-", then it is understood that we choose only one rectangle category A, and let C contain all the rest.

We also tested the effect of differing the size of the input, by using three different input sizes of $S_1 = 100$, $S_2 = 500$ and $S_3 = 1000$. The actual test instances were as follows. For each combination of $(A = R_i, B = R_j, D_k, S_l)$, we generated 20 random problem instances and ran each of them through each algorithm. There are 45 ways to choose A and B, 18 ways to choose the distribution and 3 different size classes, so the total number of instances that each algorithm solved was 48600. From the results we analysed the average and worst case performances of each algorithm.

4.2 Results

The results from all the runs are presented in tables in the appendix. In each problem instance, the average number of bins used by the algorithm was divided by the best known number of bins that were needed to pack the rectangles. This means that a score of 1.0 corresponds to a perfect performance with respect to all other algorithms, but this does not mean necessarily that the algorithm used the optimal number of bins. In each cell, the value corresponds to the average case performance, and the value in the parentheses shows the worst case performance that occurred.

4.3 The Shelf algorithms

The performance of any of the Shelf algorithms is not good enough to recommend the use of these algorithms except when fast runtime performance is needed. In online instances the Shelf algorithms can consume twice the number of bins in the worst case, and about 1.5 times on average. In offline cases, sorting by descending area and packing into multiple bins simultaneously (**-BFF**) looks like the best option, giving a 1.077 performance in the average case, but still a 1.571 times the best in the worst case.

4.4 Guillotine algorithms

On average, the Rectangle Merge was seen to improve the results in all cases, so the results from the variants without the -RM improvement were be omitted.

Also, all the different Worst Fit rules performed poorly compared to the Best Fit rules, so the Worst Fit rules were omitted in the offline case. The best average case performance in the online packing problem was obtained with the GUILLOTINE-MINAS-RM-BNF-BAF algorithm, yielding a 1.445 packing factor on average. The best worst case performance was obtained with the GUILLOTINE-LAS-RM-BNF-BSSF algorithm, which yielded a score of 1.773.

In the offline case, the Guillotine algorithms perform very well. The best average and worst case performance was obtained with the algorithm GUILLOTINE-BSSF-SAS-RM-DESCSS-BFF with scores of 1.016 and 1.111 respectively.

4.5 The MAXRECTS algorithms

The best performing algorithms are the MAXRECTS variants. When producing online packings, MAXRECTS-BSSF-BNF got a score of 1.408(1.788). If we are allowed to pack into multiple bins at once, the MAXRECTS-BSSF-BBF received a score of 1.041(1.130).

In the offline packing case, the MAXRECTS-BSSF-BBF-GLOBAL algorithm produces the ultimately best packings, with a score of 1.005(1.068). Another very good performer, and slightly faster, was the MAXRECTS-BSSF-BBF-DESCSS, which received a score of 1.009(1.087).

4.6 The SKYLINE algorithms.

The results obtained from the SKYLINE variants were quite interesting. In the online packing problem, SKYLINE-BL-WM-BNF was the best of all packers, receiving a score of 1.392(1.654). When packing into multiple bins, the SKYLINE-BL-WM-BFF got a score of 1.056(1.158), and only slightly lost to the best-performing MAXRECTS variant.

In the offline case, the best performing packer was the SKYLINE-MW-WM-BFF-DESCSS, with a score of 1.013(1.090). This is slightly better than the best offline GUILLOTINE variant, but not quite as good as the best MAXRECTS variant. It is to be noted though that the runtime performance of the SKYLINE variants is somewhat better than the MAXRECTS algorithms.

5 Conclusions and Future Work

An overall leaderboard of results is presented in the tables 21 and 22. It is clear that the MAXRECTS algorithms perform the best of all. The SKYLINE algorithms performed the best when packing is performed online only to a single bin at a time (-BNF). The GUILLOTINE variants are asymptotically faster than the MAXRECTS algorithms, but also perform slightly worse. The SHELF algorithms should only be favored if implementation simplicity is a concern.

This survey only consisted of evaluating different immediate heuristic rules. In the literature there exists several solvers that are based on meta-heuristics [14] [15], agent-based approaches [16] and iterative searching [17]. Also, publications presenting other novel heuristic approaches have appeared. These use concepts such as *corner-occupying action* and *caving degree* [18] [19], *Less Flexibility First* [20], and *Least Wasted First* [13]. In the future, it would be interesting to compare these algorithms with the best performing variants presented in this survey.

References

[1] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[2] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations* (R. E. Miller and J. W. Thatcher, eds.), pp. 85–103, Plenum Press, 1972.

[3] N. Bansal and M. Sviridenko, "New approximability and inapproximability results for 2-dimensional bin packing," in *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, (Philadelphia, PA, USA), pp. 196–203, Society for Industrial and Applied Mathematics, 2004.

[4] J. Csirik and G. J. Woeginger, "On-line packing and covering problems," in *Developments from a June 1996 seminar on Online algorithms*, (London, UK), pp. 147–177, Springer-Verlag, 1998.

[5] A. Lodi, S. Martello, and D. Vigo, "Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems," *INFORMS J. on Computing*, vol. 11, no. 4, pp. 345–357, 1999.

[6] A. Lodi, S. Martello, and D. Vigo, "Recent advances on two-dimensional bin packing problems," *Discrete Appl. Math.*, vol. 123, no. 1-3, pp. 379–396, 2002.

[7] A. Watt and F. Policarpo, *3D Games, Vol. 2: Animation and Advanced Real-Time Rendering*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[8] J. Scott, "Packing lightmaps. "http://www.blackpawn.com/texts/lightmaps/default.html"." Web.

[9] J. Ratcliff, "Blogger: John Ratcliff's Code Suppository - Texture Packing : A code snippet to compute a texture atlas. "https://www.blogger.com/comment.g?blogID=23430315&postID=2174708613887775411" ." Blog, April 2009.

[10] J. Jylänki, "Rectangle bin packing. "http://clb.demon.fi/rectangle-bin-packing"." Web.

[11] B. Chazelle, "The bottomn-left bin-packing heuristic: An efficient implementation," *IEEE Transactions on Computers*, vol. 32, no. 8, pp. 697–707, 1983.

[12] A. Naamad, D. T. Lee, and W. L. Hsu, "On the maximum empty rectangle problem," *Discrete Applied Mathematics*, vol. 8, no. 3, pp. 267 – 277, 1984.

[13] L. Wei, D. Zhang, and Q. Chen, "A least wasted first heuristic algorithm for the rectangular packing problem," *Comput. Oper. Res.*, vol. 36, no. 5, pp. 1608–1614, 2009.

[14] E. Hopper and B. C. H. Turton, "A review of the application ofmeta-heuristic algorithms to 2d strip packing problems," *Artif. Intell. Rev.*, vol. 16, no. 4, pp. 257–300, 2001.

[15] E. Hopper and B. C. H. Turton, "An empirical investigation of meta-heuristic and heuristic algorithms for a 2d packing problem," *European Journal of Operational Research*, vol. 128, pp. 34–57, 2000.

[16] S. Polyakovsky and R. M'Hallah, "An agent-based approach to the two-dimensional guillotine bin packing problem," *European Journal of Operational Research*, vol. 192, pp. 767–781, February 2009.

[17] D. Beltrán-Cano, B. Melián-Batista, and J. M. Moreno-Vega, "Solving the rectangle packing problem by an iterative hybrid heuristic," pp. 673–680, 2009.

[18] W. Huang, D. Chen, and R. Xu, "A new heuristic algorithm for rectangle packing," *Comput. Oper. Res.*, vol. 34, no. 11, pp. 3270–3280, 2007.

[19] W. Huang and D. Chen, "An efficient heuristic algorithm for rectangle-packing problem," *Simulation Modelling Practice and Theory*, vol. 15, pp. 1356–1365, November 2007.

[20] Y.-L. Wu, W. Huang, S. chung Lau, C. K. Wong, and G. H. Young, "An effective quasi-human based heuristic for solving the rectangle packing problem," *European Journal of Operational Research*, vol. 141, no. 2, pp. 341 – 358, 2002.

6 Appendix: Summary and Results

Algorithm Name	Time Complexity	Space Complexity	Input
SHELF-NF-BNF	$\Theta(n)$	$O(1)$	Online
SHELF-NF-*sort*-BNF	$\Theta(n \log n)$	$O(1)$	Offline
SHELF-NF-BFF	$\Theta(n \log n)$	$O(1)$	Online
SHELF-NF-*sort*-BFF	$\Theta(n \log n)$	$O(1)$	Offline
SHELF-FF-BNF	$O(n \log n)$	$O(n)$	Online
SHELF-FF-*sort*-BNF	$O(n \log n)$	$O(n)$	Offline
SHELF-FF-BFF	$O(n \log^2 n)$	$O(n)$	Online
SHELF-FF-*sort*-BFF	$O(n \log^2 n)$	$O(n)$	Offline
SHELF-*opt*-BNF	$O(n^2)$	$O(n)$	Online
SHELF-*opt*-*sort*-BNF	$O(n^2)$	$O(n)$	Offline
SHELF-*opt*-BFF	$O(n^2 \log n)$	$O(n)$	Online
SHELF-*opt*-*sort*-BFF	$O(n^2 \log n)$	$O(n)$	Offline
SHELF-NF-WM-BNF	$O(n^2)$	$O(n)$	Online
SHELF-NF-WM-*sort*-BNF	$O(n^2)$	$O(n)$	Offline
SHELF-NF-WM-BFF	$O(n^2 \log n)$	$O(n)$	Online
SHELF-NF-WM-*sort*-BFF	$O(n^2 \log n)$	$O(n)$	Offline
SHELF-FF-WM-BNF	$O(n^2)$	$O(n)$	Online
SHELF-FF-WM-*sort*-BNF	$O(n^2)$	$O(n)$	Offline
SHELF-FF-WM-BFF	$O(n^2 \log n)$	$O(n)$	Online
SHELF-FF-WM-*sort*-BFF	$O(n^2 \log n)$	$O(n)$	Offline
SHELF-*opt*-WM-BNF	$O(n^2)$	$O(n)$	Online
SHELF-*opt*-*sort*-WM-BNF	$O(n^2)$	$O(n)$	Offline
SHELF-*opt*-WM-BFF	$O(n^2 \log n)$	$O(n)$	Online
SHELF-*opt*-*sort*-WM-BFF	$O(n^2 \log n)$	$O(n)$	Offline
GUILLOTINE-*rect*-*split*-BNF	$O(n^2)$	$O(n)$	Online
GUILLOTINE-*rect*-*split*-*sort*-BNF	$O(n^2)$	$O(n)$	Offline
GUILLOTINE-*rect*-*split*-BFF	$O(n^2 \log n)$	$O(n)$	Online
GUILLOTINE-*rect*-*split*-*sort*-BFF	$O(n^2 \log n)$	$O(n)$	Offline
GUILLOTINE-*rect*-*split*-GLOBAL	$O(n^3)$	$O(n)$	Offline
GUILLOTINE-*rect*-*split*-RM-BNF	$O(n^3)$	$O(n)$	Online
GUILLOTINE-*rect*-*split*-RM-*sort*-BNF	$O(n^3)$	$O(n)$	Offline
GUILLOTINE-*rect*-*split*-RM-BFF	$O(n^3 \log n)$	$O(n)$	Online
GUILLOTINE-*rect*-*split*-RM-*sort*-BFF	$O(n^3 \log n)$	$O(n)$	Offline
GUILLOTINE-*rect*-*split*-RM-GLOBAL	$O(n^4)$	$O(n)$	Offline

Algorithm Name	Time Complexity	Space Complexity	Input
MAXRECTS-x-BNF	$O(\|\mathcal{F}\|^2 n)$	$O(\|\mathcal{F}\|)$	Online
MAXRECTS-x-*sort*-BNF	$O(\|\mathcal{F}\|^2 n)$	$O(\|\mathcal{F}\|)$	Offline
MAXRECTS-x-GLOBAL-BNF	$O(\|\mathcal{F}\|^2 n^2)$	$O(\|\mathcal{F}\|)$	Offline
MAXRECTS-x-BFF	$O(\|\mathcal{F}\|^2 n \log n)$	$O(\|\mathcal{F}\|)$	Online
MAXRECTS-x-*sort*-BFF	$O(\|\mathcal{F}\|^2 n \log n)$	$O(\|\mathcal{F}\|)$	Offline
MAXRECTS-x-GLOBAL-BFF	$O(\|\mathcal{F}\|^2 n^2 \log n)$	$O(\|\mathcal{F}\|)$	Offline
MAXRECTS-x-BBF	$O(\|\mathcal{F}\|^2 n^2)$	$O(\|\mathcal{F}\|)$	Online
MAXRECTS-x-*sort*-BBF	$O(\|\mathcal{F}\|^2 n^2)$	$O(\|\mathcal{F}\|)$	Offline
MAXRECTS-x-GLOBAL-BBF	$O(\|\mathcal{F}\|^2 n^3)$	$O(\|\mathcal{F}\|)$	Offline
SKYLINE-x-BNF	$O(n^2)$	$O(n)$	Online
SKYLINE-x-*sort*-BNF	$O(n^2)$	$O(n)$	Offline
SKYLINE-x-BFF	$O(n^2 \log n)$	$O(n)$	Online
SKYLINE-x-*sort*-BFF	$O(n^2 \log n)$	$O(n)$	Offline
SKYLINE-x-GLOBAL	$O(n^3)$	$O(n)$	Offline
SKYLINE-x-WM-BNF	$O(n^2)$	$O(n)$	Online
SKYLINE-x-*sort*-WM-BNF	$O(n^2)$	$O(n)$	Offline
SKYLINE-x-WM-BFF	$O(n^2 \log n)$	$O(n)$	Online
SKYLINE-x-*sort*-WM-BFF	$O(n^2 \log n)$	$O(n)$	Offline

Table 6: The final list of algorithm classes considered in the survey.

	-BNF	-BFF	-WM-BNF	-WM-BFF
SHELF-NF	1.53815(2.29835)	1.15465(1.85651)	1.4911(2.01266)	1.07304(1.21877)
SHELF-FF	1.52317(2.27066)	1.11645(1.71695)	1.4911(2.01266)	1.07304(1.21877)
SHELF-BWF	1.52382(2.27066)	1.11651(1.71695)	1.49106(2.01266)	1.073(1.21877)
SHELF-BHF	1.52552(2.27066)	1.11627(1.71695)	1.49106(2.01266)	1.073(1.21877)
SHELF-BAF	1.52395(2.27066)	1.11632(1.71695)	1.49106(2.01266)	1.073(1.21877)
SHELF-WWF	1.52738(2.27066)	1.11644(1.71695)	1.49106(2.01266)	1.073(1.21877)
SHELF-WAF	1.52718(2.27066)	1.11671(1.71695)	1.49106(2.01266)	1.073(1.21877)

Table 7: Average and worst case results for online Shelf algorithms on instances of 1000 rectangles.

	SHELF-NF	SHELF-FF	SHELF-BWF	SHELF-BHF	SHELF-BAF	SHELF-WWF	SHELF-WAF
-DESCLS-BNF	1.440 (2.033)	1.425 (1.954)	1.425 (1.954)	1.425 (1.960)	1.424 (1.955)	1.429 (1.983)	1.429 (1.979)
-DESCSS-BNF	1.455 (2.086)	1.445 (2.076)	1.446 (2.076)	1.447 (2.076)	1.447 (2.076)	1.446 (2.076)	1.446 (2.076)
-ASCDIFF-BNF	1.462 (2.004)	1.452 (1.950)	1.453 (1.950)	1.454 (1.950)	1.453 (1.950)	1.456 (1.950)	1.455 (1.950)
-DESCPERIM-BNF	1.464 (2.123)	1.454 (2.033)	1.455 (2.037)	1.455 (2.047)	1.455 (2.037)	1.457 (2.066)	1.456 (2.059)
-DESCA-BNF	1.465 (2.277)	1.455 (2.181)	1.456 (2.198)	1.456 (2.191)	1.456 (2.198)	1.456 (2.198)	1.456 (2.195)
-DESCRATIO-BNF	1.465 (2.086)	1.456 (2.076)	1.457 (2.076)	1.457 (2.076)	1.457 (2.076)	1.457 (2.076)	1.456 (2.076)
-DESCDIFF-BNF	1.474 (2.099)	1.463 (2.043)	1.463 (2.043)	1.465 (2.044)	1.463 (2.043)	1.466 (2.048)	1.466 (2.048)
-ASCLS-BNF	1.471 (2.022)	1.468 (1.988)	1.467 (1.989)	1.467 (1.989)	1.468 (1.989)	1.469 (1.998)	1.469 (1.987)
-ASCPERIM-BNF	1.497 (2.143)	1.493 (2.098)	1.493 (2.098)	1.493 (2.098)	1.493 (2.098)	1.493 (2.098)	1.493 (2.098)
-ASCA-BNF	1.516 (2.383)	1.511 (2.305)	1.511 (2.305)	1.512 (2.327)	1.512 (2.307)	1.512 (2.353)	1.512 (2.354)
-ASCRATIO-BNF	1.514 (2.248)	1.513 (2.218)	1.513 (2.218)	1.513 (2.218)	1.513 (2.218)	1.513 (2.219)	1.513 (2.219)
-ASCSS-BNF	1.515 (2.235)	1.514 (2.217)	1.514 (2.217)	1.515 (2.229)	1.515 (2.229)	1.514 (2.229)	1.514 (2.217)

Table 8: The offline SHELF-BNF variants.

	SHELF-NF	SHELF-FF	SHELF-BWF	SHELF-BHF	SHELF-BAF	SHELF-WWF	SHELF-WAF
-DESCA-BFF	1.118 (1.837)	1.077 (1.571)	1.077 (1.571)	1.077 (1.571)	1.077 (1.571)	1.077 (1.571)	1.077 (1.571)
-DESCPERIM-BFF	1.119 (1.772)	1.081 (1.589)	1.081 (1.589)	1.081 (1.589)	1.081 (1.589)	1.081 (1.589)	1.081 (1.589)
-DESCSS-BFF	1.124 (1.758)	1.083 (1.652)	1.083 (1.652)	1.083 (1.652)	1.083 (1.652)	1.083 (1.652)	1.083 (1.652)
-DESCRATIO-BFF	1.125 (1.758)	1.085 (1.652)	1.085 (1.652)	1.085 (1.652)	1.085 (1.652)	1.085 (1.652)	1.085 (1.652)
-DESCLS-BFF	1.132 (1.806)	1.106 (1.637)	1.106 (1.637)	1.106 (1.637)	1.106 (1.637)	1.107 (1.637)	1.107 (1.637)
-ASCDIFF-BFF	1.152 (1.808)	1.131 (1.672)	1.131 (1.677)	1.131 (1.659)	1.131 (1.677)	1.132 (1.664)	1.132 (1.657)
-ASCLS-BFF	1.255 (1.832)	1.254 (1.818)	1.253 (1.818)	1.253 (1.818)	1.253 (1.818)	1.254 (1.813)	1.254 (1.813)
-DESCDIFF-BFF	1.278 (1.944)	1.263 (1.885)	1.263 (1.885)	1.264 (1.885)	1.263 (1.885)	1.264 (1.885)	1.264 (1.885)
-ASCPERIM-BFF	1.386 (1.975)	1.382 (1.928)	1.382 (1.928)	1.382 (1.928)	1.382 (1.928)	1.382 (1.928)	1.382 (1.928)
-ASCA-BFF	1.453 (2.216)	1.446 (2.208)	1.446 (2.182)	1.447 (2.198)	1.446 (2.191)	1.447 (2.208)	1.447 (2.188)
-ASCRATIO-BFF	1.491 (2.166)	1.490 (2.139)	1.490 (2.139)	1.490 (2.152)	1.490 (2.152)	1.490 (2.152)	1.490 (2.139)
-ASCSS-BFF	1.498 (2.167)	1.498 (2.155)	1.498 (2.155)	1.498 (2.168)	1.498 (2.168)	1.498 (2.168)	1.498 (2.155)

Table 9: The offline SHELF-BFF variants.

	SHELF-NF	SHELF-FF	SHELF-BWF	SHELF-BHF	SHELF-BAF	SHELF-WWF	SHELF-WAF
-DESCLS-WM-BNF	1.362 (1.653)	1.362 (1.653)	1.362 (1.653)	1.362 (1.653)	1.362 (1.653)	1.362 (1.653)	1.362 (1.653)
-DESCSS-WM-BNF	1.398 (1.745)	1.398 (1.745)	1.398 (1.745)	1.398 (1.745)	1.398 (1.745)	1.398 (1.745)	1.398 (1.745)
-DESCPERIM-WM-BNF	1.400 (1.744)	1.400 (1.744)	1.400 (1.744)	1.400 (1.744)	1.400 (1.744)	1.400 (1.744)	1.400 (1.744)
-DESCA-WM-BNF	1.403 (1.749)	1.403 (1.749)	1.403 (1.749)	1.403 (1.749)	1.403 (1.749)	1.403 (1.749)	1.403 (1.749)
-DESCRATIO-WM-BNF	1.408 (1.737)	1.408 (1.737)	1.408 (1.737)	1.408 (1.737)	1.408 (1.737)	1.408 (1.737)	1.408 (1.737)
-DESCDIFF-WM-BNF	1.414 (1.678)	1.414 (1.678)	1.414 (1.678)	1.414 (1.678)	1.414 (1.678)	1.414 (1.678)	1.414 (1.678)
-ASCDIFF-WM-BNF	1.417 (1.676)	1.417 (1.676)	1.417 (1.676)	1.417 (1.676)	1.417 (1.676)	1.417 (1.676)	1.417 (1.676)
-ASCPERIM-WM-BNF	1.442 (1.800)	1.442 (1.800)	1.442 (1.800)	1.442 (1.800)	1.442 (1.800)	1.442 (1.800)	1.442 (1.800)
-ASCA-WM-BNF	1.448 (1.860)	1.448 (1.860)	1.448 (1.860)	1.448 (1.860)	1.448 (1.860)	1.448 (1.860)	1.448 (1.860)
-ASCRATIO-WM-BNF	1.455 (2.003)	1.455 (2.003)	1.455 (2.003)	1.455 (2.003)	1.455 (2.003)	1.455 (2.003)	1.455 (2.003)
-ASCSS-WM-BNF	1.456 (2.072)	1.456 (2.072)	1.456 (2.072)	1.456 (2.072)	1.456 (2.072)	1.456 (2.072)	1.456 (2.072)
-ASCLS-WM-BNF	1.457 (1.964)	1.457 (1.964)	1.457 (1.964)	1.457 (1.964)	1.457 (1.964)	1.457 (1.964)	1.457 (1.964)

Table 10: The offline SHELF-WM-BNF variants.

	SHELF-NF	SHELF-FF	SHELF-BWF	SHELF-BHF	SHELF-BAF	SHELF-WWF	SHELF-WAF
-DESCPERIM-WM-BFF	1.040 (1.177)	1.040 (1.177)	1.040 (1.177)	1.040 (1.177)	1.040 (1.177)	1.040 (1.177)	1.040 (1.177)
-DESCA-WM-BFF	1.042 (1.265)	1.042 (1.265)	1.042 (1.265)	1.042 (1.265)	1.042 (1.265)	1.042 (1.265)	1.042 (1.265)
-DESCSS-WM-BFF	1.049 (1.248)	1.049 (1.248)	1.049 (1.248)	1.049 (1.248)	1.049 (1.248)	1.049 (1.248)	1.049 (1.248)
-DESCRATIO-WM-BFF	1.051 (1.251)	1.051 (1.251)	1.051 (1.251)	1.051 (1.251)	1.051 (1.251)	1.051 (1.251)	1.051 (1.251)
-DESCLS-WM-BFF	1.072 (1.371)	1.072 (1.371)	1.072 (1.371)	1.072 (1.371)	1.072 (1.371)	1.072 (1.371)	1.072 (1.371)
-ASCDIFF-WM-BFF	1.105 (1.389)	1.105 (1.389)	1.105 (1.384)	1.105 (1.384)	1.105 (1.384)	1.105 (1.384)	1.105 (1.384)
-DESCDIFF-WM-BFF	1.194 (1.507)	1.194 (1.507)	1.194 (1.507)	1.194 (1.507)	1.194 (1.507)	1.194 (1.507)	1.194 (1.507)
-ASCLS-WM-BFF	1.246 (1.780)	1.246 (1.780)	1.246 (1.788)	1.246 (1.788)	1.246 (1.788)	1.246 (1.788)	1.246 (1.788)
-ASCPERIM-WM-BFF	1.320 (1.648)	1.320 (1.648)	1.320 (1.648)	1.320 (1.648)	1.320 (1.648)	1.320 (1.648)	1.320 (1.648)
-ASCA-WM-BFF	1.364 (1.693)	1.364 (1.693)	1.364 (1.693)	1.364 (1.693)	1.364 (1.693)	1.364 (1.693)	1.364 (1.693)
-ASCRATIO-WM-BFF	1.401 (1.732)	1.401 (1.732)	1.401 (1.732)	1.401 (1.732)	1.401 (1.732)	1.401 (1.732)	1.401 (1.732)
-ASCSS-WM-BFF	1.408 (1.744)	1.408 (1.744)	1.408 (1.744)	1.408 (1.744)	1.408 (1.744)	1.408 (1.744)	1.408 (1.744)

Table 11: The offline SHELF-WM-BFF variants.

	-BAF	-BLSF	-BSSF	-WAF	-WLSF	-WSSF
GUILLOTINE-MINAS-RM-BNF	1.445 (2.301)	1.447 (2.132)	1.454 (2.849)	1.511 (2.892)	1.510 (3.385)	1.517 (2.803)
GUILLOTINE-MINAS-BNF	1.446 (2.320)	1.449 (2.156)	1.455 (2.859)	1.513 (2.915)	1.512 (3.411)	1.520 (2.864)
GUILLOTINE-LAS-RM-BNF	1.467 (1.836)	1.473 (2.004)	1.483 (1.773)	1.558 (3.569)	1.561 (2.587)	1.552 (3.636)
GUILLOTINE-LAS-BNF	1.468 (1.852)	1.474 (2.049)	1.484 (1.783)	1.561 (3.620)	1.564 (2.619)	1.554 (3.696)
GUILLOTINE-SLAS-RM-BNF	1.468 (2.709)	1.474 (2.765)	1.461 (2.837)	1.544 (3.388)	1.517 (3.386)	1.560 (3.524)
GUILLOTINE-SLAS-BNF	1.470 (2.727)	1.475 (2.771)	1.462 (2.853)	1.547 (3.416)	1.519 (3.412)	1.563 (3.562)
GUILLOTINE-SAS-RM-BNF	1.614 (3.433)	1.643 (3.460)	1.594 (3.357)	1.690 (3.519)	1.644 (3.616)	1.739 (3.655)
GUILLOTINE-SAS-BNF	1.615 (3.451)	1.645 (3.467)	1.596 (3.377)	1.693 (3.537)	1.648 (3.661)	1.743 (3.673)
GUILLOTINE-LLAS-RM-BNF	1.616 (2.861)	1.634 (2.891)	1.616 (2.791)	1.715 (3.493)	1.676 (3.010)	1.713 (3.536)
GUILLOTINE-LLAS-BNF	1.617 (2.872)	1.635 (2.894)	1.617 (2.801)	1.717 (3.558)	1.678 (3.018)	1.715 (3.588)
GUILLOTINE-MAXAS-RM-BNF	1.634 (3.060)	1.658 (3.327)	1.623 (2.814)	1.732 (3.249)	1.689 (3.030)	1.751 (3.576)
GUILLOTINE-MAXAS-BNF	1.635 (3.070)	1.659 (3.330)	1.625 (2.830)	1.735 (3.266)	1.692 (3.037)	1.754 (3.586)

Table 12: The online GUILLOTINE variants.

	-DESC SS -BFF	-DESC RATIO -BFF	-DESCA -BFF	-DESC PERIM -BFF	-GLOBAL	-DESC LS -BFF
GUILLOTINE-BSSF-SAS-RM	1.016 (1.111)	1.017 (1.111)	1.019 (1.128)	1.021 (1.148)	1.036 (1.142)	1.043 (1.396)
GUILLOTINE-BAF-SAS-RM	1.017 (1.112)	1.018 (1.112)	1.020 (1.135)	1.022 (1.159)	1.020 (1.135)	1.044 (1.396)
GUILLOTINE-BSSF-LLAS-RM	1.019 (1.125)	1.019 (1.125)	1.023 (1.144)	1.026 (1.172)	1.035 (1.141)	1.043 (1.396)
GUILLOTINE-BSSF-MAXAS-RM	1.019 (1.125)	1.020 (1.125)	1.023 (1.144)	1.025 (1.172)	1.035 (1.141)	1.043 (1.396)
GUILLOTINE-BLSF-SAS-RM	1.019 (1.136)	1.020 (1.136)	1.021 (1.149)	1.023 (1.173)	1.024 (1.164)	1.046 (1.396)
GUILLOTINE-BAF-LLAS-RM	1.020 (1.125)	1.021 (1.125)	1.025 (1.144)	1.028 (1.172)	1.025 (1.145)	1.048 (1.396)
GUILLOTINE-BAF-MAXAS-RM	1.021 (1.127)	1.022 (1.127)	1.024 (1.144)	1.027 (1.172)	1.024 (1.145)	1.047 (1.396)
GUILLOTINE-BLSF-LLAS-RM	1.023 (1.125)	1.024 (1.125)	1.026 (1.151)	1.030 (1.173)	1.032 (1.181)	1.052 (1.396)
GUILLOTINE-BLSF-MAXAS-RM	1.024 (1.127)	1.025 (1.127)	1.026 (1.151)	1.028 (1.173)	1.030 (1.181)	1.052 (1.396)
GUILLOTINE-BSSF-SLAS-RM	1.026 (1.155)	1.026 (1.155)	1.023 (1.163)	1.023 (1.160)	1.020 (1.156)	1.046 (1.396)
GUILLOTINE-BSSF-MINAS-RM	1.027 (1.168)	1.028 (1.168)	1.024 (1.173)	1.025 (1.169)	1.020 (1.156)	1.047 (1.396)
GUILLOTINE-BSSF-LAS-RM	1.027 (1.311)	1.028 (1.317)	1.031 (1.309)	1.035 (1.329)	1.019 (1.133)	1.048 (1.396)
GUILLOTINE-BLSF-SLAS-RM	1.029 (1.210)	1.030 (1.215)	1.027 (1.223)	1.027 (1.222)	1.040 (1.330)	1.045 (1.396)
GUILLOTINE-BAF-SLAS-RM	1.029 (1.215)	1.029 (1.211)	1.027 (1.227)	1.027 (1.222)	1.027 (1.223)	1.045 (1.396)
GUILLOTINE-BLSF-MINAS-RM	1.037 (1.371)	1.038 (1.370)	1.035 (1.340)	1.036 (1.332)	1.044 (1.412)	1.046 (1.396)
GUILLOTINE-BAF-MINAS-RM	1.037 (1.372)	1.037 (1.371)	1.034 (1.340)	1.036 (1.332)	1.035 (1.339)	1.046 (1.396)
GUILLOTINE-BAF-LAS-RM	1.048 (1.479)	1.049 (1.484)	1.050 (1.495)	1.052 (1.484)	1.050 (1.494)	1.051 (1.396)
GUILLOTINE-BLSF-LAS-RM	1.048 (1.479)	1.049 (1.484)	1.049 (1.495)	1.051 (1.484)	1.049 (1.471)	1.052 (1.396)

Table 13: The offline GUILLOTINE variants.

	-ASC DIFF -BFF	-DESC DIFF -BFF	-ASCLS -BFF	-ASC PERIM -BFF	-ASCA - BFF	-ASC RATIO -BFF	-ASCSS -BFF
GUILLOTINE-BSSF-SAS-RM	1.084 (1.344)	1.158 (1.465)	1.222 (1.506)	1.294 (1.567)	1.324 (1.593)	1.385 (1.739)	1.391 (1.739)
GUILLOTINE-BAF-SAS-RM	1.086 (1.353)	1.161 (1.472)	1.225 (1.509)	1.299 (1.560)	1.333 (1.610)	1.399 (1.769)	1.405 (1.776)
GUILLOTINE-BSSF-LLAS-RM	1.087 (1.348)	1.161 (1.467)	1.221 (1.497)	1.295 (1.558)	1.336 (1.625)	1.407 (1.776)	1.413 (1.786)
GUILLOTINE-BSSF-MAXAS-RM	1.086 (1.348)	1.161 (1.468)	1.227 (1.498)	1.298 (1.558)	1.336 (1.612)	1.402 (1.771)	1.408 (1.782)
GUILLOTINE-BLSF-SAS-RM	1.088 (1.364)	1.164 (1.478)	1.235 (1.522)	1.312 (1.595)	1.341 (1.617)	1.406 (1.776)	1.412 (1.783)
GUILLOTINE-BAF-LLAS-RM	1.092 (1.371)	1.166 (1.473)	1.230 (1.513)	1.302 (1.560)	1.345 (1.633)	1.420 (1.807)	1.426 (1.819)
GUILLOTINE-BAF-MAXAS-RM	1.092 (1.372)	1.165 (1.472)	1.236 (1.527)	1.306 (1.560)	1.346 (1.625)	1.414 (1.795)	1.420 (1.806)
GUILLOTINE-BLSF-LLAS-RM	1.095 (1.388)	1.170 (1.477)	1.239 (1.541)	1.311 (1.605)	1.348 (1.633)	1.424 (1.821)	1.430 (1.831)
GUILLOTINE-BLSF-MAXAS-RM	1.095 (1.388)	1.169 (1.478)	1.245 (1.533)	1.317 (1.593)	1.349 (1.630)	1.418 (1.803)	1.424 (1.811)
GUILLOTINE-BSSF-SLAS-RM	1.084 (1.332)	1.161 (1.477)	1.166 (1.571)	1.286 (1.601)	1.308 (1.598)	1.362 (1.695)	1.367 (1.701)
GUILLOTINE-BSSF-MINAS-RM	1.084 (1.332)	1.165 (1.480)	1.158 (1.565)	1.282 (1.597)	1.307 (1.599)	1.363 (1.696)	1.368 (1.701)
GUILLOTINE-BSSF-LAS-RM	1.090 (1.428)	1.173 (1.503)	1.160 (1.503)	1.282 (1.599)	1.326 (1.615)	1.387 (1.726)	1.392 (1.732)
GUILLOTINE-BLSF-SLAS-RM	1.091 (1.340)	1.171 (1.551)	1.171 (1.514)	1.308 (1.636)	1.327 (1.634)	1.378 (1.714)	1.383 (1.719)
GUILLOTINE-BAF-SLAS-RM	1.090 (1.341)	1.170 (1.548)	1.168 (1.494)	1.304 (1.635)	1.324 (1.638)	1.374 (1.709)	1.379 (1.714)
GUILLOTINE-BLSF-MINAS-RM	1.095 (1.457)	1.184 (1.660)	1.158 (1.497)	1.306 (1.717)	1.333 (1.731)	1.388 (1.742)	1.393 (1.742)
GUILLOTINE-BAF-MINAS-RM	1.094 (1.458)	1.184 (1.660)	1.157 (1.490)	1.304 (1.717)	1.331 (1.731)	1.386 (1.742)	1.391 (1.742)
GUILLOTINE-BAF-LAS-RM	1.106 (1.562)	1.203 (1.697)	1.164 (1.489)	1.315 (1.708)	1.354 (1.736)	1.411 (1.810)	1.416 (1.810)
GUILLOTINE-BLSF-LAS-RM	1.107 (1.567)	1.203 (1.697)	1.164 (1.499)	1.314 (1.708)	1.353 (1.736)	1.410 (1.810)	1.416 (1.810)

Table 14: The offline GUILLOTINE variants.

	-BNF	-BFF	-BBF
MAXRECTS-BSSF	1.408 (1.788)	1.047 (1.134)	1.041 (1.130)
MAXRECTS-BAF	1.420 (1.817)	1.047 (1.134)	1.043 (1.132)
MAXRECTS-BLSF	1.436 (1.708)	1.051 (1.155)	1.052 (1.181)
MAXRECTS-CP	1.411 (1.669)	1.049 (1.142)	1.062 (1.206)
MAXRECTS-BL	1.388 (1.648)	1.051 (1.157)	1.280 (1.486)

Table 15: The online MAXRECTS variants.

	-DESC SS	-DESC RATIO	-DESCA	-DESC PERIM	-GLOBAL	-DESC LS
MAXRECTS-BL-BFF	1.008 (1.091)	1.009 (1.091)	1.013 (1.120)	1.015 (1.125)		1.041 (1.396)
MAXRECTS-BSSF-BBF	1.009 (1.087)	1.010 (1.087)	1.010 (1.106)	1.012 (1.111)	1.005 (1.068)	1.035 (1.396)
MAXRECTS-BSSF-BFF	1.009 (1.087)	1.010 (1.087)	1.012 (1.106)	1.014 (1.111)		1.040 (1.396)
MAXRECTS-CP-BFF	1.009 (1.087)	1.009 (1.087)	1.012 (1.109)	1.014 (1.111)		1.042 (1.396)
MAXRECTS-BAF-BFF	1.009 (1.088)	1.010 (1.088)	1.012 (1.108)	1.014 (1.111)		1.041 (1.396)
MAXRECTS-BLSF-BFF	1.010 (1.086)	1.011 (1.090)	1.014 (1.102)	1.017 (1.119)		1.045 (1.396)
MAXRECTS-BAF-BBF	1.010 (1.088)	1.010 (1.088)	1.011 (1.107)	1.012 (1.111)	1.010 (1.083)	1.036 (1.396)
MAXRECTS-BLSF-BBF	1.011 (1.087)	1.012 (1.094)	1.014 (1.103)	1.016 (1.117)	1.011 (1.089)	1.042 (1.396)
MAXRECTS-CP-BBF	1.011 (1.116)	1.012 (1.122)	1.012 (1.100)	1.014 (1.109)	1.012 (1.121)	1.040 (1.395)
MAXRECTS-BL-BBF	1.030 (1.186)	1.031 (1.183)	1.062 (1.161)	1.096 (1.335)	1.480 (1.862)	1.198 (1.592)
MAXRECTS-BL-BNF	1.360 (1.696)	1.360 (1.697)	1.365 (1.692)	1.355 (1.686)	1.343 (1.666)	1.329 (1.585)
MAXRECTS-CP-BNF	1.374 (1.717)	1.375 (1.715)	1.375 (1.697)	1.358 (1.680)	1.010 (1.120)	1.325 (1.577)
MAXRECTS-BSSF-BNF	1.384 (1.732)	1.385 (1.732)	1.389 (1.729)	1.367 (1.676)	1.005 (1.068)	1.311 (1.556)
MAXRECTS-BAF-BNF	1.389 (1.737)	1.390 (1.736)	1.398 (1.734)	1.389 (1.790)	1.010 (1.083)	1.316 (1.557)
MAXRECTS-BLSF-BNF	1.390 (1.738)	1.391 (1.739)	1.396 (1.706)	1.397 (1.767)	1.011 (1.089)	1.338 (1.602)

Table 16: The offline MAXRECTS variants.

	-ASC DIFF	-DESC DIFF	-ASCLS	-ASC PERIM	-ASCA	-ASC RATIO	-ASCSS
MAXRECTS-BL-BFF	1.050 (1.311)	1.152 (1.466)	1.125 (1.483)	1.237 (1.524)	1.290 (1.565)	1.350 (1.694)	1.355 (1.694)
MAXRECTS-BSSF-BBF	1.043 (1.317)	1.147 (1.456)	1.122 (1.488)	1.235 (1.532)	1.279 (1.554)	1.346 (1.675)	1.351 (1.681)
MAXRECTS-BSSF-BFF	1.052 (1.323)	1.150 (1.456)	1.127 (1.488)	1.239 (1.531)	1.282 (1.554)	1.348 (1.677)	1.353 (1.682)
MAXRECTS-CP-BFF	1.051 (1.324)	1.151 (1.460)	1.126 (1.474)	1.236 (1.516)	1.285 (1.547)	1.345 (1.670)	1.350 (1.675)
MAXRECTS-BAF-BFF	1.052 (1.323)	1.151 (1.465)	1.130 (1.491)	1.243 (1.536)	1.286 (1.571)	1.350 (1.673)	1.355 (1.678)
MAXRECTS-BLSF-BFF	1.056 (1.338)	1.154 (1.459)	1.138 (1.526)	1.249 (1.571)	1.295 (1.581)	1.359 (1.681)	1.364 (1.686)
MAXRECTS-BAF-BBF	1.049 (1.323)	1.150 (1.463)	1.126 (1.490)	1.239 (1.538)	1.284 (1.577)	1.349 (1.673)	1.354 (1.677)
MAXRECTS-BLSF-BBF	1.055 (1.340)	1.157 (1.463)	1.135 (1.533)	1.245 (1.577)	1.292 (1.581)	1.358 (1.680)	1.363 (1.686)
MAXRECTS-CP-BBF	1.069 (1.410)	1.158 (1.477)	1.161 (1.498)	1.261 (1.642)	1.302 (1.675)	1.353 (1.728)	1.359 (1.728)
MAXRECTS-BL-BBF	1.176 (1.472)	1.322 (1.731)	1.410 (1.920)	1.459 (1.848)	1.462 (1.835)	1.480 (1.868)	1.487 (1.874)
MAXRECTS-BL-BNF	1.363 (1.677)	1.342 (1.691)	1.338 (1.590)	1.352 (1.687)	1.369 (1.694)	1.374 (1.710)	1.373 (1.709)
MAXRECTS-CP-BNF	1.375 (1.686)	1.368 (1.697)	1.337 (1.585)	1.357 (1.685)	1.382 (1.698)	1.392 (1.740)	1.392 (1.744)
MAXRECTS-BSSF-BNF	1.411 (1.848)	1.374 (1.694)	1.327 (1.565)	1.352 (1.672)	1.397 (1.746)	1.418 (1.775)	1.418 (1.776)
MAXRECTS-BAF-BNF	1.413 (1.887)	1.379 (1.698)	1.335 (1.569)	1.361 (1.675)	1.408 (1.756)	1.449 (1.912)	1.449 (1.912)
MAXRECTS-BLSF-BNF	1.432 (1.865)	1.392 (1.727)	1.354 (1.608)	1.382 (1.705)	1.407 (1.715)	1.446 (1.871)	1.445 (1.871)

Table 17: The offline MAXRECTS variants.

	-BNF	-BFF
SKYLINE-BL-WM	1.392 (1.654)	1.056 (1.158)
SKYLINE-BL	1.398 (1.658)	1.069 (1.235)
SKYLINE-MW-WM	1.413 (1.659)	1.054 (1.141)
SKYLINE-MW	1.416 (1.751)	1.064 (1.187)

Table 18: The online SKYLINE variants.

	SKYLINE-BL-WM-BNF	SKYLINE-MW-WM-BNF
-DESCLS	1.329 (1.583)	1.330 (1.586)
-ASCLS	1.337 (1.588)	1.342 (1.596)
-DESCDIFF	1.344 (1.691)	1.348 (1.689)
-ASCPERIM	1.355 (1.689)	1.362 (1.688)
-DESCPERIM	1.358 (1.687)	1.367 (1.683)
-DESCSS	1.361 (1.697)	1.369 (1.705)
-DESCRATIO	1.361 (1.698)	1.369 (1.706)
-DESCA	1.369 (1.697)	1.381 (1.698)
-ASCA	1.373 (1.700)	1.389 (1.708)
-ASCDIFF	1.362 (1.675)	1.390 (1.704)
-ASCSS	1.374 (1.710)	1.407 (1.772)
-ASCRATIO	1.374 (1.710)	1.408 (1.772)

Table 19: The offline SKYLINE-BNF variants.

	SKYLINE-BL-WM-BFF	SKYLINE-MW-WM-BFF
-DESCSS	1.013 (1.094)	1.013 (1.090)
-DESCRATIO	1.013 (1.094)	1.013 (1.090)
-DESCA	1.017 (1.123)	1.017 (1.112)
-DESCPERIM	1.019 (1.125)	1.019 (1.111)
-DESCLS	1.041 (1.396)	1.041 (1.396)
-ASCDIFF	1.051 (1.313)	1.054 (1.323)
-ASCLS	1.126 (1.485)	1.127 (1.489)
-DESCDIFF	1.154 (1.466)	1.154 (1.456)
-ASCPERIM	1.239 (1.530)	1.238 (1.526)
-ASCA	1.294 (1.571)	1.289 (1.568)
-ASCRATIO	1.352 (1.681)	1.352 (1.675)
-ASCSS	1.356 (1.683)	1.357 (1.680)

Table 20: The offline SKYLINE-BFF variants.

MAXRECTS-BSSF-BBF	1.04063 (1.13026)
MAXRECTS-BAF-BBF	1.04256 (1.13231)
MAXRECTS-BSSF-BFF	1.04669 (1.13411)
MAXRECTS-BAF-BFF	1.04728 (1.13409)
MAXRECTS-CP-BFF	1.04911 (1.142)
MAXRECTS-BL-BFF	1.05065 (1.15721)
MAXRECTS-BLSF-BFF	1.05144 (1.15477)
MAXRECTS-BLSF-BBF	1.05181 (1.18114)
SKYLINE-MW-WM-BFF	1.05391 (1.14136)
SKYLINE-BL-WM-BFF	1.05569 (1.15824)
MAXRECTS-CP-BBF	1.06154 (1.20645)
GUILLOTINE-BSSF-SAS-RM-BFF	1.06222 (1.15781)
GUILLOTINE-BSSF-SAS-BFF	1.06273 (1.15781)
GUILLOTINE-BSSF-SLAS-RM-BFF	1.06275 (1.20376)
GUILLOTINE-WLSF-SAS-RM-BFF	1.06349 (1.16027)
GUILLOTINE-BSSF-SLAS-BFF	1.0639 (1.20679)
GUILLOTINE-WLSF-SLAS-RM-BFF	1.06401 (1.20328)
SKYLINE-MW-BFF	1.06403 (1.18656)
GUILLOTINE-WLSF-SAS-BFF	1.06408 (1.16039)
GUILLOTINE-BSSF-MINAS-RM-BFF	1.0642 (1.23088)
GUILLOTINE-WLSF-SLAS-BFF	1.06509 (1.20681)
GUILLOTINE-BSSF-MINAS-BFF	1.06529 (1.23549)
GUILLOTINE-WLSF-MINAS-RM-BFF	1.06532 (1.23439)
GUILLOTINE-BAF-SAS-RM-BFF	1.06549 (1.17989)
GUILLOTINE-BAF-SAS-BFF	1.06593 (1.17996)
GUILLOTINE-WLSF-MINAS-BFF	1.06651 (1.23544)
GUILLOTINE-WAF-SAS-RM-BFF	1.06654 (1.17853)
GUILLOTINE-WAF-SAS-BFF	1.06705 (1.17834)
GUILLOTINE-BLSF-SAS-RM-BFF	1.06725 (1.1804)
GUILLOTINE-BLSF-SAS-BFF	1.06761 (1.1804)
GUILLOTINE-BAF-SLAS-RM-BFF	1.06873 (1.30024)
GUILLOTINE-WSSF-SAS-RM-BFF	1.06882 (1.18171)
GUILLOTINE-WSSF-SAS-BFF	1.06932 (1.18171)
SKYLINE-BL-BFF	1.06948 (1.23466)
GUILLOTINE-BSSF-MAXAS-RM-BFF	1.06963 (1.17194)
GUILLOTINE-BSSF-MAXAS-BFF	1.06984 (1.17162)
GUILLOTINE-BLSF-SLAS-RM-BFF	1.07008 (1.29906)

Table 21: Overall online variants.

MAXRECTS-BSSF-GLOBAL-BBF	1.00466 (1.06773)
MAXRECTS-BSSF-GLOBAL-BNF	1.00466 (1.06773)
MAXRECTS-BL-DESCSS-BFF	1.00849 (1.0909)
MAXRECTS-CP-DESCSS-BFF	1.00858 (1.08664)
MAXRECTS-BSSF-DESCSS-BBF	1.00898 (1.08718)
MAXRECTS-BL-DESCRATIO-BFF	1.00907 (1.0909)
MAXRECTS-CP-DESCRATIO-BFF	1.00912 (1.08664)
MAXRECTS-BSSF-DESCSS-BFF	1.00922 (1.08699)
MAXRECTS-BAF-DESCSS-BFF	1.00949 (1.08754)
MAXRECTS-BSSF-DESCRATIO-BBF	1.00952 (1.08718)
MAXRECTS-BAF-DESCSS-BBF	1.00957 (1.08754)
MAXRECTS-BSSF-DESCRATIO-BFF	1.0098 (1.08699)
MAXRECTS-BAF-DESCRATIO-BFF	1.01006 (1.08754)
MAXRECTS-BAF-GLOBAL-BBF	1.01013 (1.08258)
MAXRECTS-BAF-GLOBAL-BNF	1.01013 (1.08258)
MAXRECTS-BAF-DESCRATIO-BBF	1.01021 (1.08754)
MAXRECTS-CP-GLOBAL-BNF	1.01045 (1.1199)
MAXRECTS-BSSF-DESCA-BBF	1.01045 (1.10625)
MAXRECTS-BLSF-DESCSS-BFF	1.01046 (1.08601)
MAXRECTS-BLSF-DESCSS-BBF	1.0107 (1.08674)
MAXRECTS-BLSF-DESCRATIO-BFF	1.0111 (1.09038)
MAXRECTS-BLSF-GLOBAL-BNF	1.01121 (1.08929)
MAXRECTS-BAF-DESCA-BBF	1.01127 (1.10651)
MAXRECTS-BLSF-GLOBAL-BBF	1.01129 (1.08929)
MAXRECTS-CP-DESCSS-BBF	1.01138 (1.11591)
MAXRECTS-BLSF-DESCRATIO-BBF	1.0115 (1.09397)
MAXRECTS-BSSF-DESCA-BFF	1.01163 (1.10553)
MAXRECTS-CP-DESCA-BFF	1.01167 (1.10924)
MAXRECTS-BAF-DESCA-BFF	1.01182 (1.10771)
MAXRECTS-CP-DESCRATIO-BBF	1.01193 (1.1219)
MAXRECTS-CP-DESCA-BBF	1.01221 (1.09965)
MAXRECTS-BSSF-DESCPERIM-BBF	1.01237 (1.11109)
MAXRECTS-CP-GLOBAL-BBF	1.01243 (1.12094)
MAXRECTS-BAF-DESCPERIM-BBF	1.01246 (1.11109)
MAXRECTS-BL-DESCA-BFF	1.01269 (1.12007)
SKYLINE-MW-DESCSS-WM-BFF	1.01281 (1.08979)

Table 22: Overall offline variants.