

A fast and scalable bottom-left-fill algorithm to solve nesting problems using a semi-discrete representation

Sahar Chehrazad^a, Dirk Roose^a and Tony Wauters^b

^a*KU Leuven, Department of Computer Science, campus Leuven, Belgium*

^b*KU Leuven, Department of Computer Science, Technologicampus Gent, Belgium*

ARTICLE INFO

Keywords:

Cutting

Nesting problems

Semi-discrete representation

Bottom-left-fill algorithm

Sweep-line algorithm

ABSTRACT

We present a fast algorithm to solve nesting problems based on a semi-discrete representation of both the 2D non-convex pieces and the strip. The pieces and the strip are represented by a set of equidistant vertical line segments. The discretization algorithm uses a sweep-line method and applies minimal extensions to the line segments of a piece to ensure that non-overlapping placement of the segments, representing two pieces, cannot cause overlap of the original pieces. We implemented a bottom-left-fill greedy placement procedure, using an optimised ordering of the segments overlap tests. The C++ implementation of our algorithm uses appropriate data structures that allow fast execution. It executes the bottom-left-fill algorithm for typical ESICUP data sets in a few milliseconds, even when rotation of the pieces is considered, and thus provides a suitable ‘building block’ for integration in metaheuristics. Moreover, we show that the algorithm scales well when the number of pieces increases.


1. Introduction

Nesting problems, a branch of cutting and packing problems, are important for many industries, e.g. textile, sheet metal, leather, glass and paper industries, see [Bennell and Oliveira \(2008\)](#) and also in additive manufacturing. In this paper the goal is to place 2D, possibly non-convex, pieces without overlap on a strip, a rectangular sheet with a fixed width and a variable length, while minimizing the used length of the strip. This is known as the irregular strip packing problem, which is a branch of ‘open dimension’ problems, see [Wascher, Haußner and Schumann \(2007\)](#).

For a classification of nesting problems and for an overview of the many solution strategies we refer to review papers such as [Bennell and Oliveira \(2008\)](#); [Gomes \(2013\)](#); [Leao, Toledo, Oliveira, Carravilla and Alvarez-Valdés \(2020\)](#). The solution strategies vary from a combination of specific heuristics to using a general-purpose MIP solver, based on a mathematical model of the problem. Here we only highlight two aspects of the problem and its solution, namely a) continuous vs. discrete representation of the pieces and the strip and b) the quality of the solution vs. the run-time and the scalability of the algorithms.

Many methods use a continuous representation of the pieces and the strip. We refer to the use of the no-fit polygon, see [Amaro Junior, Pinheiro, Saraiva and Pinheiro \(2014\)](#); [Pinheiro, Amaro Junior and Saraiva \(2016\)](#), or the use of ‘direct trigonometry’ as in [Burke, Hellier, Kendall and Whitwell \(2006\)](#). Several methods combine a continuous representation of the pieces with a discretized strip, as for example in [Burke et al. \(2006\)](#) and in the dotted board model, see for example [Toledo, Carravilla, Ribeiro, Oliveira and Gomes \(2013\)](#). A two-dimensional discretization of both the pieces and the strip is used in e.g. [Sato, Tsuzuki, Martins and Gomes \(2016b\)](#); [Sato, Martins, Gomes and Tsuzuki \(2019\)](#). Of course, any discretization or approximation of the geometry reduces the potential quality of the solution, but the effect on the actually achieved solution quality can be negative or positive, since nearly all methods use heuristics because the nesting problem is NP-hard. The discretization effect will be smaller if the pieces and/or strip are discretized in only one direction, instead of in both directions, as mentioned in [Burke et al. \(2006\)](#). However, such a semi-discretization has not received much attention so far.

Due to the iterative nature of most heuristic methods, the solution quality and the run-time are conflicting aspects. It is important that the basic operation in the iteration, e.g. a bottom-left placement of pieces in a given order or testing

 Sahar.chehrazad@kuleuven.be (S. Chehrazad); Dirk.Roose@kuleuven.be (D. Roose); Tony.Wauters@kuleuven.be (T. Wauters)

 <https://people.cs.kuleuven.be/~dirk.roose/Site/Home.html> (D. Roose)

the constraints in a MIP method, is executed in a short time, since this allows, within a given computing budget, to perform more iterations, allowing potentially a better solution. The run-time of an algorithm not only depends on the number of operations performed, but also on the data structures and the data access pattern. Indeed, due to the availability of several pipelined functional units on a single core to perform floating point operations, with increasing length of the pipeline, and the availability of SIMD vector instructions, the gap between the time to perform a (floating point) operation and the time to access data from (main) memory has increased over time and is substantial in current processors. This so called ‘DRAM gap’ is alleviated by the presence of several levels of cache memory, with different access times, due to the different latency and bandwidth to transfer data to the next level in the hierarchy. To achieve high performance, the computer code must exploit the functionality of this memory hierarchy. When the data used in consecutive steps of the algorithm is stored in consecutive memory locations, the data can often be fetched from the cache, i.e. a high cache hit ratio is achieved. For more information we refer to Hager and Wellein (2011). Further, the computational complexity as a function of the number of pieces determines the scalability of the method, but this aspect is often not discussed in detail.

The aim of this paper is to explore the potential of a semi-discretization of both the pieces and the strip to achieve high performance, and thus a short run-time, for a basic placement method that can be used as a basic operation or ‘building block’ in a heuristic method to efficiently solve the nesting problem. We also aim to assess the scalability of the approach. We expect that semi-discretization can lead to a good compromise between accuracy and computational complexity. Semi-discretization of both pieces and strip transforms the problem of placing pieces without overlap into the problem of placing line segments without overlap. We will see that the core of the algorithm consists of simple comparisons of floating point numbers that are stored in regular data structures with a rather regular data access pattern, which allows to achieve high performance on current processors.

Only a few papers have explored the use of semi-discretization of pieces and/or strip.

In Ma and Liu (2007), a semi-discrete representation discretizes the pieces and the strip by a number of vertical rectangles of the same width. The lower and upper bound of each rectangle is equal to the lower and upper bound of the part of the piece lying in that rectangle. Each vertical rectangle of the board represents empty space on the board. A feasible position for each piece is determined by checking whether the empty rectangles of the strip can contain all the rectangles of the piece. Even though the proposed representation guarantees that overlap can never happen, it results in wasted area.

Akunuru and Babu (2013) proposed another semi-discrete representation for the pieces and the strip. Here the discretization does not consist of vertical rectangles but consists of a set of equidistant vertical line segments. The semi-discretization of a piece is computed by traversing the circumference of the piece and intersecting the edges of the piece at x -coordinates that are multiples of the distance between the vertical lines. The intersection points are stored in a table and the vertical line segments are derived from this table. They propose an extension procedure to avoid overlap of pieces during placement. However, the extension rules do not cover all cases, for example, when a whole edge of the piece lies between two of the vertical equidistant lines. Also, the implementation of the placement heuristic (a bottom-left-fill algorithm) is not sufficiently detailed to be able to re-implement the algorithm.

In Leao, Toledo, Oliveira and Carravilla (2015), the semi-discrete representation discretizes the pieces and the board along the y -axis while keeping the x -axis continuous. A mixed integer programming model (MIP) is used to solve the nesting problem. The NFP is used to guarantee that the pieces do not overlap while placing them in the board. The results indicate that for data sets with more than ten pieces the execution time is prohibitively high.

Since the strip is also semi-discretized and represented by vertical line segments, the number of potential locations to place a piece is infinite, due to its continuous y -axis property, as the ‘variable shift’ approach in Burke et al. (2006). To test whether a piece can be placed in a certain position, we must test whether all line segments representing an (extended) piece do not overlap the line segments of the (partially filled) strip. Placements of pieces in the strip using this representation can result in a better solution than using a grid-based method, and requires only simple calculations when compared to the no-fit polygon method or other direct representations of the pieces. Note that, while computing the NFP of two convex pieces is simple and cheap, the NFP of non-convex pieces is more complex and time-consuming, especially when the problem specification allows rotation of the pieces, and is known to have numerical problems, see Wauters, Uyttersprot and Esprit (2016).

In this paper we show that a bottom-left-fill greedy placement heuristic, even when considering several rotation angles of the pieces, can be executed in a few milliseconds on a single core of a current processor, for classical benchmark data sets, when appropriate data structures are used on which the required calculations can be executed efficiently.

Of course, the simple bottom-left-fill heuristic cannot achieve high quality solutions to the nesting problem, but many approaches in the literature use metaheuristics on top of this greedy heuristic, e.g. Gomes and Oliveira (2002); Burke et al. (2006); Ma and Liu (2007); Akunuru and Babu (2013); Amaro Junior et al. (2014); Pinheiro et al. (2016); Sato et al. (2019). These metaheuristics can be applied regardless of the representation of the pieces and the strip.

The remainder of the paper is organized as follows. Section 2 presents the precise formulation of the problem and the data structures in C++ that we use to achieve high performance. In section 3, the construction of the semi-discrete representation using a sweep-line algorithm along with the extension algorithm is explained in detail. Section 4 describes the bottom-left-fill placement heuristic and how rotations are handled. In section 5 computational results on benchmark problems from the literature are presented, providing insight in the performance of the method, i.e. the quality of the solution, the run-time and the scalability w.r.t. the number of pieces. Finally, in section 6 we draw some conclusions and direction for future research.

2. Problem formulation and data structures

As mentioned in section 1, this paper deals with the irregular strip packing problem, a branch of ‘open dimension’ problems, see Wascher et al. (2007). The aim is to place two-dimensional, possibly non-convex, polygonal pieces without overlap on a rectangular strip, while minimizing the used length of the strip. In case rotation of pieces is allowed, the placement of each piece is checked with a discrete set of rotation angles. The basic operation in most of the placement heuristics is the placement of a piece in a partially filled strip according to some criterion. We present a bottom-left-fill placement heuristic, where both strip and pieces are semi-discretized. We discretize the pieces and the strip along the horizontal x -axis. Each vertical line with x -coordinate $x_i = i \times R$, is called a resolution line and the constant distance R between the vertical lines indicates the resolution. The semi-discrete representation of the pieces and the strip consists of a set of vertical line segments on the resolution lines.

We assume the bottom left corner of the axis-aligned bounding box of the piece has (local) coordinates $(x, y) = (0, 0)$. Denote the bottom and top endpoints of the j -th line segment on the i -th resolution line by respectively $(x_i, b_{i,j})$ and $(x_i, t_{i,j})$ with $x_i = i \times R$, $i = 0, 1, 2, \dots$. The number of line segments on a resolution line varies; for a convex piece, there is only one line segment on each resolution line. The partially filled semi-discretized strip consists of vertical line segments, indicating space occupied by already placed pieces. We assume that the bottom left corner of the strip has (strip) coordinates $(0, 0)$. Denote the bottom and top endpoints of the l -th line segment on the k -th resolution line by $(x_k, b_{k,l}^s)$ and $(x_k, t_{k,l}^s)$ with $x_k = k \times R$, $k = 0, 1, 2, \dots$, where superscript s refers to the strip.

In the bottom-left-fill heuristic a piece is placed in the left-most and bottom-most position in the partially filled strip without overlap. For a semi-discretized piece and strip this requires to find a ‘translation vector’ $t = (x_t, y_t)$ with $x_t = m \cdot R$ such that the y -intervals $(b_{i,j} + y_t, t_{i,j} + y_t)$ and $(b_{i+m,l}^s, t_{i+m,l}^s)$, representing y -coordinates of the endpoints of the line segments do not overlap $\forall i = 0, 1, 2, \dots, \forall j, \forall l$, such that m is minimal and, for that m , y_t is minimal. This requires a number of tests, involving only the y -coordinates of the line segments of both the piece and partially filled strip, see Fig. 9. Ordering the line segments of both the piece and the partially filled strip in each resolution line, such that $b_{i,j+1} > t_{i,j}$ and $b_{k,l+1}^s > t_{k,l}^s$, limits the number of tests since overlap between $(b_{i,j} + y_t, t_{i,j} + y_t)$ and $(b_{i+m,q}^s, t_{i+m,q}^s)$ with $q > l$ is not possible if $t_{i,j} + y_t < t_{i+m,l}^s$.

The required data are thus the ordered set of y -intervals $(b_{i,j}, t_{i,j})$ for the piece and the ordered set of intervals $(b_{k,l}^s, t_{k,l}^s)$ for the partially filled strip. To select appropriate data structures, we take into account that *a*) for each translation vector, all the line segments of a piece can be involved in the overlap tests and *b*) the run-time of a program is nowadays mainly determined by the data access pattern, rather than by the number of operations, due to the difference in time to load data from the memory hierarchy and to execute floating point operations. In order to optimize the cache hit ratio, the y -intervals $(b_{i,j}, t_{i,j})$, $j = 0, 1, 2, \dots, i = 0, 1, 2, \dots$, representing the y -coordinates of the line segments of the piece, should be stored in consecutive memory locations. Hence a suitable data structure in C++ is an array of arrays or a vector of vectors. Considering the fact that initially the number of segments on each resolution line of the piece and strip is not known, we use vectors. Indeed, vector elements are placed in contiguous storage, as in arrays, so that they can be accessed and traversed efficiently using iterators. In addition, vectors can be initialized with a certain length, but they automatically re-scale their memory as needed. However, vectors do not reallocate memory each time an element is added. Instead, vector containers may allocate some extra storage to accommodate for possible growth in length. Accessing an arbitrary element of the vector is very efficient, as in arrays, and also inserting an element at the end of a vector is done in constant time, see Cplusplus.com (2020). For similar reasons, the ordered set of intervals

$(b_{k,l}^s, t_{k,l}^s)$ representing the y -coordinates of the line segments of the partially filled strip are also stored in a vector of vectors.

3. Semi-discretization of a piece

3.1. Sweep-line algorithm

The semi-discrete representation of a polygonal piece is constructed using a sweep-line algorithm, see de Berg, Cheong, van Kreveld and Overmars (2008). To efficiently determine the line segments lying inside the piece on the resolution lines at $x_i = i \times R$, $i = 0, 1, 2, \dots$, the resolution lines are considered from left to right (we assume that the left-most vertex has x -coordinate $x = 0$). To visualize the algorithm one can assume a vertical ‘sweep-line’, traversing the piece, and thus also the resolution lines, from left to right. When the edges of the piece, that intersect the resolution line under consideration, are determined and sorted in the right order, the y -intervals mentioned in the previous section (y -coordinates of the endpoints of the line segments) can easily be computed and stored in increasing order in a C++ vector `Ypoints`. Therefore, the sweep-line algorithm maintains a list of the ‘active’ edges of the piece, i.e. the edges that intersect the resolution line under consideration, in the vector `ActiveEdges`, sorted according to increasing y -coordinate. The data structure `ActiveEdges` must be updated when the sweep-line passes a vertex of the piece. Therefore, the vector `Events` contains the x -coordinates of the vertices sorted in increasing order.

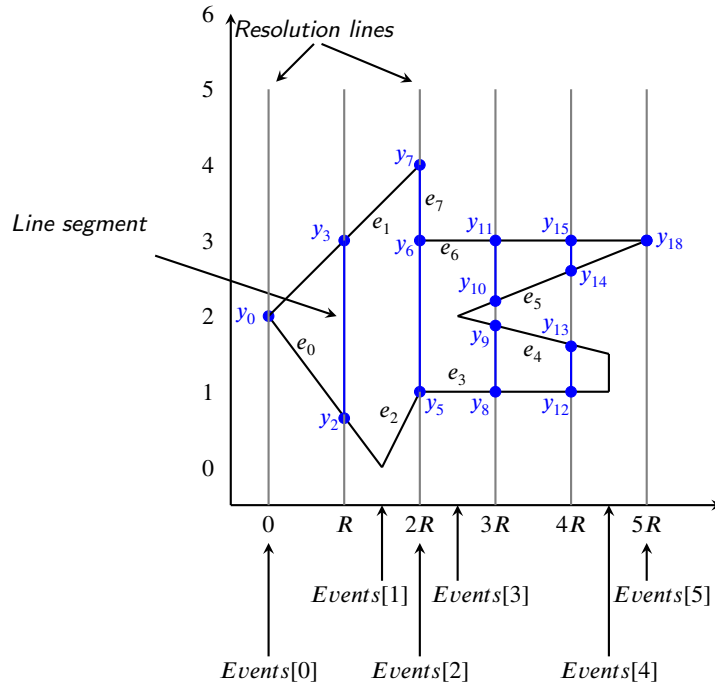


Figure 1: Components of sweep-line algorithm.

Traversing the y -coordinates in `Ypoints` in increasing order gives the y -intervals $(b_{i,j}, t_{i,j})$ representing the y -coordinates of the endpoints of the line segments of the piece on the resolution line at $x_i = x \times R$.

We assign a label P to each line segment, representing the position of the segment in the semi-discretized piece. This label will be used during placement to allow that pieces touch each other, see section 4. Therefore, (the y -coordinates of) each segment, together with the label, are stored as a tuple $(b_{i,j}, t_{i,j}, P)$. Considering the position of the segments in a piece gives three different values for P . The position label of a segment on a resolution line between the current and the next event is M (Middle). On a resolution line coinciding with an event, a segment has the label R (Right), if the inside of the piece lies to the right of the segment, has the label L (Left), if the inside of the piece lies to the left of the segment.

The tuples at $x_i = i \times R$, $i = 0, 1, 2, \dots$, are saved in a vector $\text{Piece}[i]$, which is an element of the vector Piece . Hence at the end of the sweep-line algorithm, Piece contains all vectors of tuples of the piece. More information on the details of the sweep-line algorithm can be found in Appendix.A.

For example in Fig. 2, after processing $\text{Events}[m]$, ActiveEdges contains the information of four edges, i.e. e_0 , e_1 , e_2 and e_3 , and y -coordinates of the intersection points at $x_i = i \times R$, stored in Ypoints , are y_0, \dots, y_3 , and the tuples are (y_0, y_1, M) and (y_2, y_3, M) . We also illustrate some special cases when the resolution line coincides with an event. The y -coordinates of the intersection points at $x_i = i \times R$, coinciding with $\text{Events}[m]$, are y_4, y_5, y_6 and y_6 and the tuples are (y_4, y_5, M) and (y_6, y_6, R) . The y -coordinates of intersection points at $x_i = i \times R$, coinciding with $\text{Events}[m+1]$, are y_7, y_8, y_8 and y_9 and the tuples are (y_7, y_8, M) and (y_8, y_9, M) .

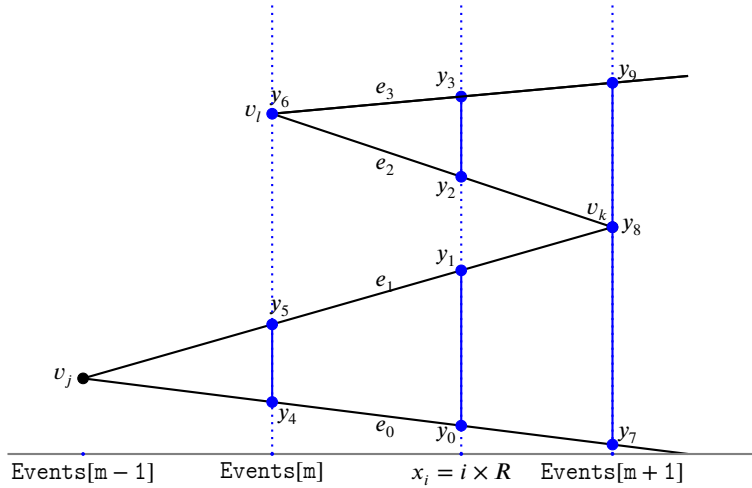


Figure 2: Computing the intervals of a piece at three different x -coordinates; intervals (y_0, y_1, M) and (y_2, y_3, M) at $x_i = i \times R$, intervals (y_4, y_5, M) and (y_6, y_6, R) at x_i coinciding with $\text{Events}[m]$ and intervals (y_7, y_8, M) and (y_8, y_9, M) at x_i coinciding with $\text{Events}[m+1]$.

3.2. Extension algorithm

The purpose of the extension algorithm is to guarantee that if there is no overlap between the line segments of two semi-discretized pieces, the corresponding original pieces do not overlap. The input of the extension algorithm is Piece , with $\text{Piece}[i]$ a vector of tuples (y_1, y_2, P) , where (y_1, y_2) is an interval corresponding to a line segment of the semi-discretization of a piece on resolution line at $x_i = i \times R$. In the remainder of the text we will merely use the word ‘interval’, even if we refer to the corresponding tuple or even to the corresponding line segment.

Some information of the piece is lost when a vertex lies between two neighboring resolution lines. For each convex vertex (x_v, y_v) located between two resolution lines, extra intervals, called extension intervals, will be added to the set of intervals representing the piece in one or both neighboring resolution lines. Suppose $v = (x_v, y_v)$ is a convex vertex with $x_i = i \times R < x_v < x_{i+1} = (i+1) \times R$. In order to insert the extension intervals in $\text{Piece}[i]$ and $\text{Piece}[i+1]$, the union of the intervals which are already in these vectors and the extension intervals are computed. Hence, if the extension interval already exists in the vector, it will not be added again. For each convex vertex both adjacent edges are considered. For each adjacent edge e , we distinguish two cases.

1. If edge e intersects a neighboring resolution line, the extension will happen on both of the two neighboring resolution lines. Let (x_i, y_A) be the intersection point of the edge and the neighboring resolution line, see Fig. 3. The extension interval on the resolution line which intersects e is equal to (y_A, y_v, P) , with $P = R$ if e intersects the left resolution line and $P = L$ if e intersects the right resolution line. On the other neighboring resolution line, which is not intersected by e , the extension interval is equal to (y_v, y_v, P) with $P = R$ if e intersects the right resolution line and $P = L$ if e intersects the left resolution line. In case the other adjacent edge \tilde{e} intersects the same resolution line, as in Fig. 3a, a similar calculation would lead to extension intervals on both resolution lines that overlap with existing intervals and extension intervals and hence must not be added. However, in case

\tilde{e} , the other edge adjacent to vertex (x_v, y_v) , intersects the other neighboring resolution line, i.e. at (x_{i+1}, y_B) , a similar calculation results in the extension interval (y_B, y_v, P) , see Fig. 3b.

2. If the edge intersects none of the neighboring resolution lines, i.e. the whole edge is placed between resolution lines, the extension interval on both resolution lines is the orthogonal projection of the edge and is equal to (y_{v_1}, y_{v_2}, P) , with $P = R$ on the left resolution line, and $P = L$ on the right resolution line, see Figs. 3c and 4. If the extension interval overlaps with an existing interval with the third element of the tuple (i.e. the position of the existing interval) equal to M no update is needed.

When a non-convex vertex lies between two resolution lines, no extension is needed, see Fig. 5, since by applying the extension algorithm for all convex vertices, without extension due to non-convex vertices, overlap with another piece cannot happen.

Besides dealing with vertices that lie between resolution lines, there is another situation where extension can be necessary. It is possible that placement of another piece causes overlap of the pieces without overlapping line segments of the semi-discretization of the pieces, see Fig. 6a. This can happen when there are gaps between the y -intervals on neighboring resolution lines, if these intervals correspond to line segments lying between the same edges of the piece. In these gaps intervals of the other piece could be placed. This situation is avoided by extending such intervals so that intervals on neighbouring resolution lines always overlap, see Fig. 6b. The default option in our implementation is to perform this extension only for intervals of length zero (y_v, y_v, P) (i.e. a vertex lying on a resolution line), see Figs. 6c and 6d, if $b \geq y_v$ or $t \leq y_v$ where b and t denote the bottom and top endpoints of the interval on the neighboring resolution line. Indeed, non-overlapping intervals in neighboring resolution lines, corresponding to interior line segments, only occur for a piece with a specific geometry, i.e. when two consecutive edges have both a large positive or both a large negative slope, and semi-discretized with a large R . In such a case this extension can also be activated for the interior resolution lines or R can be decreased. The latter will also improve the accuracy of the semi-discrete representation.

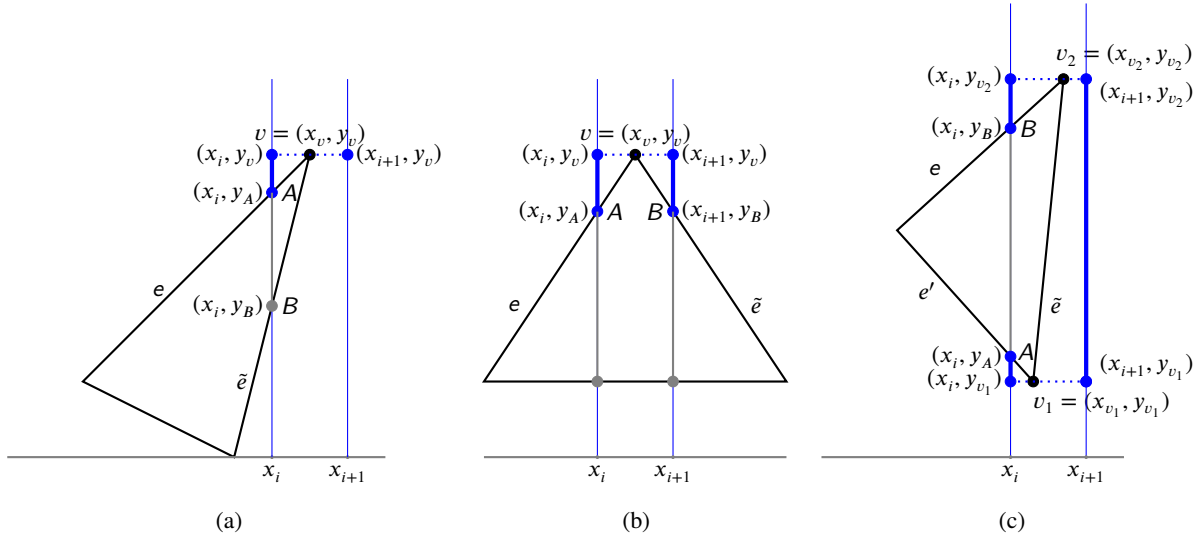


Figure 3: Extension intervals on each neighboring resolution line for a convex vertex $v = (x_v, y_v)$. In 3a, the extension intervals due to edge \tilde{e} , i.e. (y_B, y_v, R) at x_i and (y_v, y_v, L) at x_{i+1} , overlap with existing (extension) intervals and will not be added. In 3b, the extension intervals due to edge \tilde{e} are (y_v, y_v, R) at x_i and (y_B, y_v, L) at x_{i+1} . However, (y_v, y_v, R) at x_i and (y_v, y_v, L) at x_{i+1} overlap with existing extension intervals. In 3c, edges e and e' intersect the resolution line at x_i and edge \tilde{e} does not intersect any resolution line. The extension intervals due to \tilde{e} are (y_{v_1}, y_{v_2}, R) at x_i and (y_{v_1}, y_{v_2}, L) at x_{i+1} . However (y_{v_1}, y_A, R) , (y_A, y_B, M) and (y_B, y_{v_2}, R) already exist at x_i and (y_{v_2}, y_{v_2}, L) and (y_{v_1}, y_{v_1}, L) already exist at x_{i+1} .

3.3. Final data structure

In a last step, tuples corresponding to touching intervals and with the same third element are joined, i.e. when for two neighboring tuples the second element of one tuple is equal to the first element of the other tuple and they have an

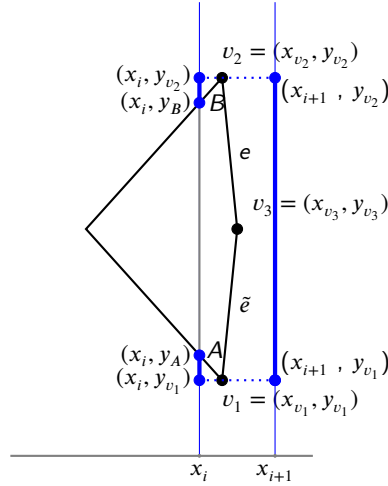


Figure 4: Extension intervals when none of the edges e and \tilde{e} , adjacent to a convex vertex v_3 , intersect resolution lines.

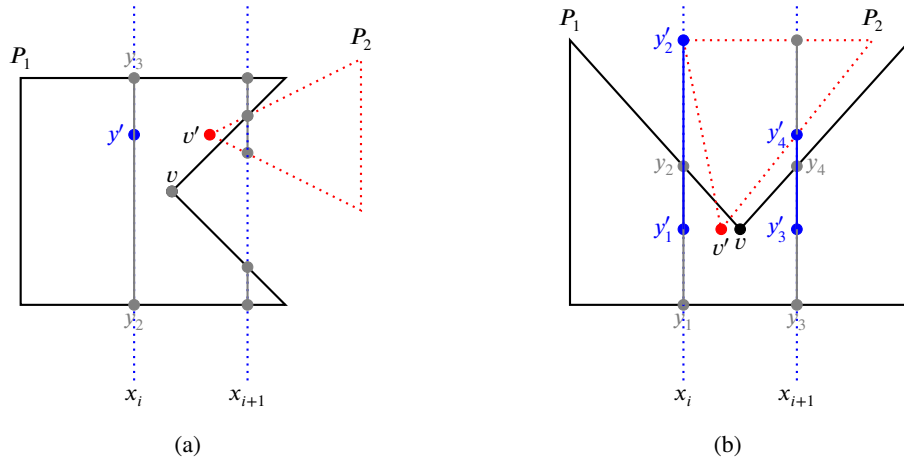


Figure 5: No extension is needed for a non-convex vertex v of piece P_1 . 5a: placing semi-discretized piece P_2 , with convex vertex v' inside P_1 between resolution lines at x_i and x_{i+1} causes overlap of extension interval (y', y', R) of P_2 with (y_2, y_3, M) . 5b: placing semi-discretized P_2 inside P_1 between resolution lines at x_i and x_{i+1} causes overlap of the extension intervals (y'_1, y'_2, R) and (y'_3, y'_4, L) with (y_1, y_2, M) and (y_3, y_4, M) .

identical third element. The final semi-discretization of the piece, i.e. all tuples corresponding to intervals representing line segments of the piece, are stored in a vector of vectors of tuples, called *Piece*. Fig. 7b shows a semi-discretized piece with $R = 1$ and the data structure *Piece* is given by:

$$\text{Piece} = \left[\begin{array}{l} [(y_0, y_0, R)], [(y_1, y_2, R), (y_2, y_3, M)], [(y_4, y_5, L), (y_5, y_6, M), (y_6, y_7, L)], \\ [(y_8, y_9, M), (y_{10}, y_{11}, M)], [(y_{12}, y_{13}, M), (y_{14}, y_{15}, M)], [(y_{16}, y_{17}, L), (y_{18}, y_{18}, L)] \end{array} \right]$$

3.4. Resolution

The extension region, which is equal to the sum of the regions of the triangles and trapezoids enclosed between the resolution lines and the edges of the piece, see Fig. 8, is considered as the discretization error. The discretization error depends on R ; it increases with increasing R and consequently, a larger part of the strip is not available for placement of other pieces, see Fig. 8b. However, the computational cost of placing pieces on the strip decreases with increasing R as the number of line segments representing a piece decreases and consequently the amount of computations required for checking whether a piece can be placed in a position decreases, see Fig. 8a. Therefore, we will use several values

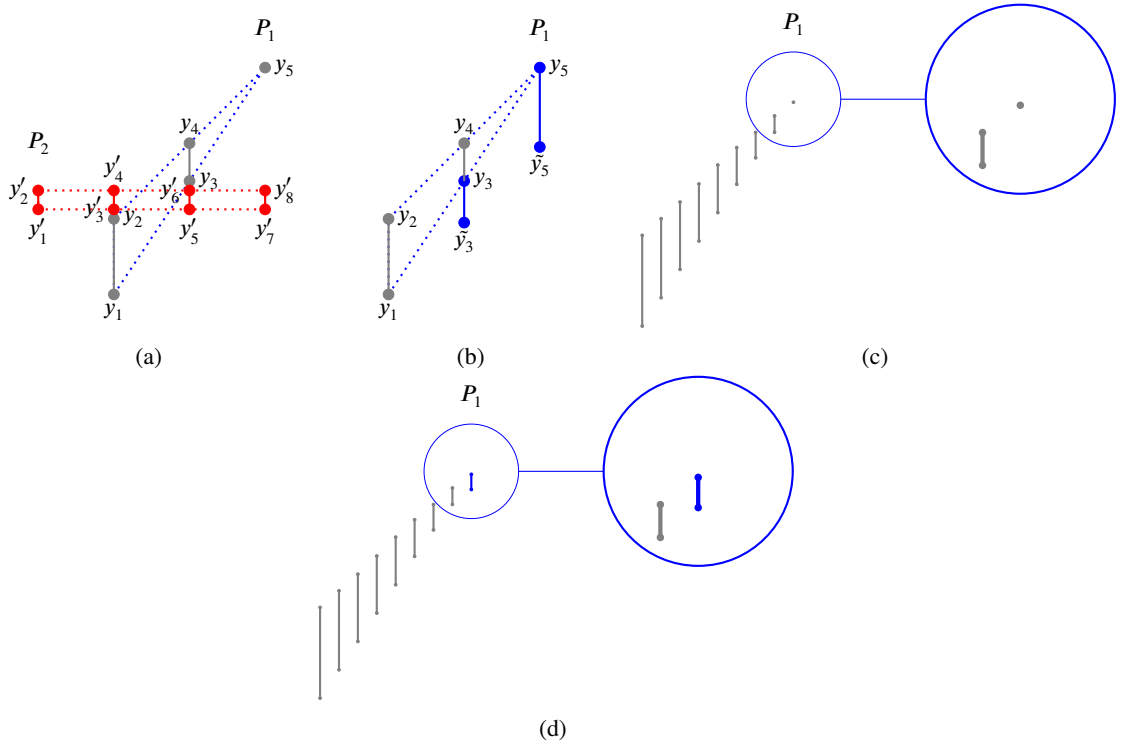


Figure 6: In 6a, there are gaps between the y -intervals of P_1 on neighboring resolution lines in which the intervals of P_2 can be placed. In 6b, the problem is solved by extending the non-overlapping intervals to remove the gaps. In 6c, a smaller R -value solves the problem in interior intervals. However the last interval (of length zero) has no overlap with the interval in the consecutive resolution line. In 6d, the problem is solved by extension of the last interval.

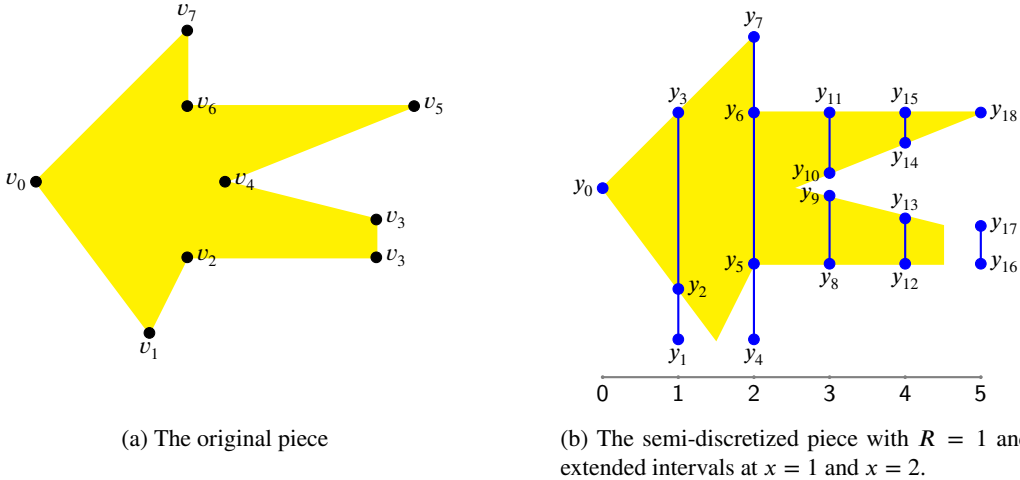


Figure 7: An example of discretization and extension of a piece.

for R in our tests. The base resolution R_b is set to $\max\{P_e, P_p/n_e\}$, where P_e is the smallest x -projection of all non-vertical edges of all pieces, P_p is the x -projection of the smallest piece and n_e is the number of edges of the smallest piece. This leads to a good accuracy of the representation of the pieces by line segments as in most of the cases P_e is larger than P_p/n_e and this leads to at least one line segment per edge. However, for data sets with near vertical edges, $R = P_e$ would be very small, leading to an excessive computational cost. Therefore, in this case, we choose P_p/n_e as

the base resolution. R -values smaller than R_b are also considered, to improve the quality of the solution by reducing the extension regions and the gaps between pieces.

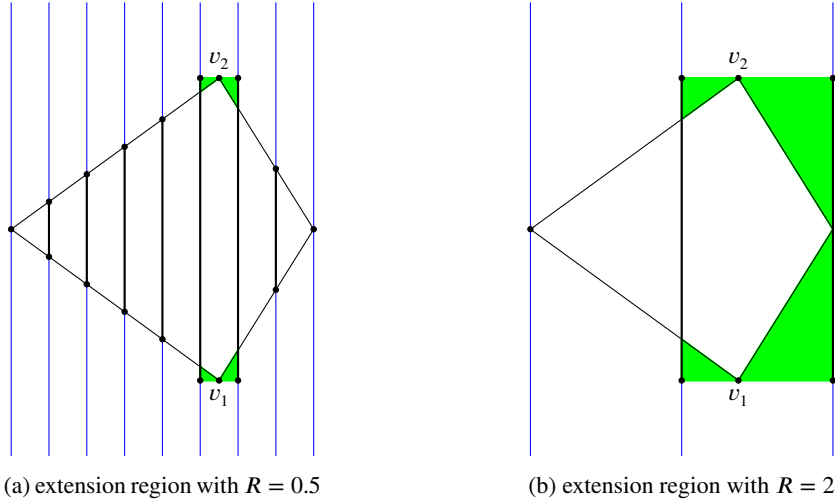


Figure 8: The extension region for different R -values. It increases with increasing R .

4. Placement

We now describe the bottom-left-fill placement of the semi-discretized pieces. The strip is a rectangle with a fixed width and a variable length, semi-discretized along the x -axis with the same distance R between the resolution lines as for the pieces. The parts of each resolution line, i.e. at $x_i = i \times R$, $i = 0, 1, 2, \dots$, that are occupied by line segments of already placed pieces are stored in a vector of vectors of tuples, i.e. a similar data structure as used to represent the pieces. However, we will merely use the word ‘interval’ instead of ‘tuple’ (or ‘line segment’) both for the pieces and the strip. The pieces are sorted, e.g. according to decreasing bounding box area, discretized and subsequently placed in the strip. If several copies of the same piece are placed consecutively, the piece is discretized only once.

To describe the placement of a semi-discretized piece, we refine the notations introduced in section 2.

- The y -coordinates $b_{i,j}$ and $t_{i,j}$ in tuple $(b_{i,j}, t_{i,j}, P_{i,j})$ are called the local coordinates, where $b_{i,j}$ and $t_{i,j}$ denote respectively the bottom and the top endpoint of the j -th interval on the i -th resolution line of the piece. The label $P_{i,j}$ indicates the position of the interval in the piece, see section 3.1. The bottom left corner of the axis-aligned bounding box of the piece has (local) coordinates $(x, y) = (0, 0)$.
- The bottom-left corner of the strip has (strip) coordinates $(x, y) = (0, 0)$. The partially filled semi-discretized strip consists of tuples $(b_{k,l}^s, t_{k,l}^s, P_{k,l}^s)$, indicating space occupied by already placed pieces, with $b_{k,l}^s$ and $t_{k,l}^s$ denoting respectively the bottom and the top endpoint of the l -th interval on the k -th resolution line, with superscript s referring to the strip. The label $P_{k,l}^s$ indicates the position of the interval in the union of the already placed intervals.
- When an interval of a piece is (tentatively) placed in the strip, the coordinates of its endpoints in the strip are called the strip coordinates. The local coordinates are transformed into the strip coordinates by adding a ‘translation vector’ $t = (x_t, y_t)$ with $x_t = m \cdot R$, $m = 0, 1, 2, \dots$, such that m is minimal and, for that m , y_t is minimal. Hence t indicates the position of the bottom left corner of the axis-aligned bounding box of the piece in the strip. According to the bottom-left strategy we initialize the translation vector as $t = (0, 0)$.

Due to the semi-discretization, placing a piece in the strip without overlap corresponds to finding a translation vector t such that the intervals of the piece do not overlap with the filled intervals in the strip (unless the labels $P_{i,j}$ and $P_{k,l}^s$ allow overlap). This requires tests with the strip coordinates of the endpoints of the intervals $(b_{i,j} + y_t, t_{i,j} + y_t)$, $i = 0, 1, 2, \dots$, of the piece and the coordinates of the endpoints of the filled intervals in resolution lines $(b_{i+m,l}^s, t_{i+m,l}^s)$ $\forall j, \forall l$. Fig. 9 illustrates the placement algorithm.

We process the intervals of the piece in a given order, starting with the leftmost interval. For every interval, we test whether the interval can be placed in the strip with the current translation vector as follows. An interval with the label $P_{i,j} = M$ can be placed if there is no overlap with an already filled interval. An interval with label $P_{i,j} = R/L$ can be placed if there is no overlap with an already filled intervals in the strip or if there is an overlap with an already filled interval with label $P_{i+m,l}^S = L/R$.

For a given translation vector, it is not necessary to perform the tests for all intervals to decide that the current translation vector does not allow placement without overlap. Indeed, as soon as an interval cannot be placed as mentioned above, we can update the translation vector t . Hence, we proceed as follows.

1. If the interval can be placed under the above mentioned conditions, we consider the next interval.
2. If the interval cannot be placed, we try to place it in the current resolution line, by shifting it repeatedly upwards and testing whether it overlaps with the next filled interval on the current resolution line.
 - (a) If there is no overlap, the interval can be placed on the current resolution line. We compute the required shift in the y -direction and add it to y_t to update the translation vector and we consider the next interval.
 - (b) If the interval cannot be placed on the current resolution line, we break the loop over the intervals, we return to the first (leftmost) interval and we update the translation vector to $t = (x_t + R, 0)$, i.e. x_t is set to the x -value of the next resolution line and y_t is set to the bottom point on that resolution line, i.e. $y_t = 0$.

If we have performed the loop over all intervals, we distinguish two cases.

1. If the translation vector has not been updated during the whole loop over the intervals, then the translation vector allows to place all the intervals on the strip. The piece is then placed and the filled intervals of the strip are updated as follows.
 - (a) If an interval is placed without overlap with the already filled intervals, the interval is added to the strip and the label of the tuple is equal to that of the piece interval.
 - (b) If an interval with label equal to R/L is placed on an already filled interval with label equal to L/R , the label of the overlapping interval in the strip is updated to M , i.e. no more intervals can be placed on it.
2. If the translation vector has been updated, the loop over the intervals must be repeated with the updated translation vector, to ensure that indeed all intervals of the piece can be placed with this translation vector.

The order of checking the intervals of the piece during placement can have an impact on the total number of tests and the execution time. Indeed, each test with an interval can result in an update of the translation vector. Consecutive piece intervals often have endpoints with nearly equal y -coordinates and the same holds for the consecutive strip intervals. Checking intervals in the order of increasing x -coordinates is therefore often not optimal, since this ordering only slowly reveals global information about the possibility to place the piece. Therefore, we check the intervals of a piece covering $L + 1$ resolution lines in the order $\{0, L, L/2, L/4, 3L/4, \dots\}$ which quickly gives information about the possible placement of intervals located at the beginning, end, and middle of the piece. Therefore this can accelerate the decision about the placement of the piece in that position of the strip.

The placement can be done more efficiently when there are several copies of the same piece in the data set. When such a piece has been placed in the strip with translation vector $t = (x_t, y_t)$, the search for finding an optimal position for the next copy of that piece starts with this translation vector instead of with $t = (0, 0)$.

As an example, consider the placement of eight copies of a piece from the data set ‘Shirts’, called P_0, \dots, P_7 , see Fig. 10. Each piece covers 13 resolution lines, each with 1 interval, denoted by $I_{i,0}$, except for the 4-th resolution line where there are two intervals $I_{3,0}$ and $I_{3,1}$ (with different position labels). The placement of P_0 requires zero checks because there are no filled intervals in the strip yet. The placement of P_1 starts with translation vector $t = (0, 0)$, but for this translation the interval $I_{0,0}$, represented by tuple $(b_{0,0}, t_{0,0}, R)$, of P_1 overlaps with interval $I_{0,0}^S$, represented by tuple $(b_{0,0}^S, t_{0,0}^S, R)$ in the strip, therefore, it is shifted up over a distance $L_{0,0}^S$, see Fig. 10. Since there is no overlap, the next interval of P_1 is checked. The same amount of checks is required for intervals $I_{12,0}, I_{6,0}, I_{9,0}, I_{1,0}, I_{4,0}, I_{7,0}, I_{10,0}, I_{2,0}, I_{5,0}, I_{8,0}$ and $I_{11,0}$. Intervals $I_{3,0}$ and $I_{3,1}$ are checked separately and require one check with each filled intervals $I_{3,0}^S$ and $I_{3,1}^S$ of the strip. The translation vector is updated for $I_{0,0}, I_{3,0}, I_{10,0}$ and $I_{11,0}$. With the last updated translation

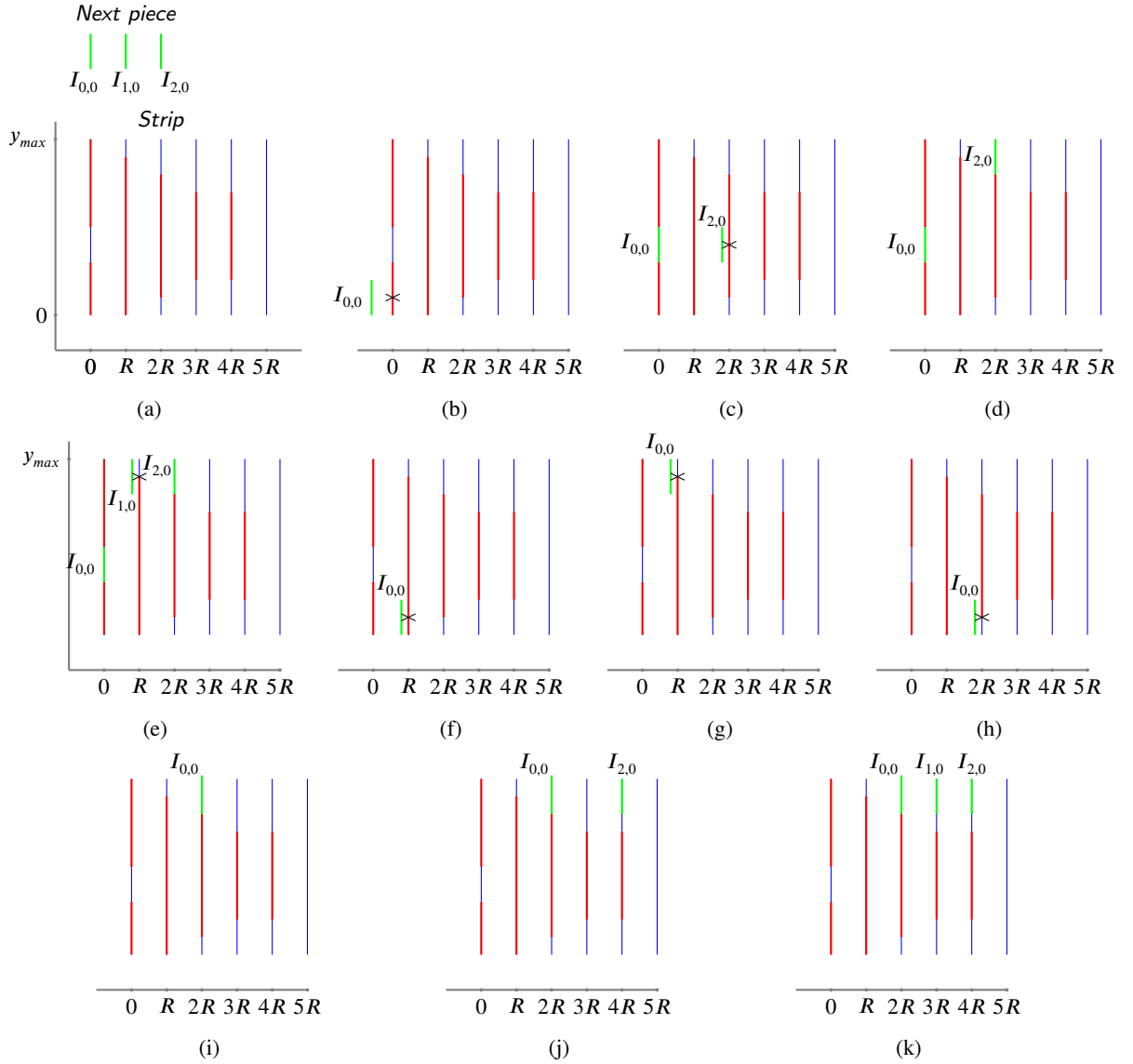


Figure 9: Placement algorithm: 9a represents the strip with already placed pieces where red indicates filled intervals. In 9b, interval $I_{0,0}$ cannot be placed with $t = (0, 0)$ so we shift it upwards. In 9c, $I_{0,0}$ is placed at $x = 0$ with $t = (0, 1)$, but $I_{2,0}$ cannot be placed. Therefore, in 9d, we shift $I_{2,0}$ upwards, t is updated to $(0, 2)$. In 9e, $I_{1,0}$ cannot be placed. We shift the intervals to the right and update t . In 9f and 9g we cannot place $I_{0,0}$ at $x = R$. Therefore, We again shift the intervals to the right and update t . In 9i, 9j and 9k, $I_{0,0}$, $I_{2,0}$ and $I_{1,0}$ can be placed at respectively $x = 2R$, $x = 4R$ and $x = 3R$ with $t = (2R, 2)$.

vector, all 14 intervals of P_1 are checked again and the same amount of checks are required. In total 32 checks are required for placing P_1 . The same number of interval checks and updates of the translation vector are needed to place P_2 and P_3 , as the placements of P_2 and P_3 start with the translation vectors of P_1 and P_2 , respectively. However, 32 and 64 extra checks are also required to search for the indexes of the filled intervals. Placement of P_4 starts by checking interval $I_{0,0}$ with the translation vector of P_3 . Only one check is done in resolution line at $x = 0$ of the strip with 3 extra checks to find the index of the last filled interval. For each interval represented by tuple $(b_{i,j}, t_{i,j}, P_{i,j})$ with length $L_{i,j}$, no placement check is done in the interval $(y_{max} - L_{i,j}, y_{max}]$ of the resolution lines of the strip. Then the search starts in the next resolution line at $x = R$, i.e. one shift to the right. On resolution lines at $x = R$ to $x = 11R$ of the strip 48 checks are done, i.e. 8 checks on resolution line $x = 3R$ and 4 checks on each of other resolution lines. On resolution

Table 1

Number of translation vector updates and interval checks for placing eight copies of a piece.

Piece	Updates in translation vector	Shift right	Total number of checks
P_0	0	0	0
P_1	4	0	32
P_2	4	0	64
P_3	4	0	96
P_4	12	12	53
P_5	4	0	36
P_6	4	0	70
P_7	4	0	104

line at $x = 12R$, the last check is done for interval $I_{0,0}$ of P_4 . As the next vectors of the strip are empty, no checks are required for the other 13 intervals of P_4 . In total 53 checks are required to place this piece. The placement of P_5 is similar to the placement of P_2 , however, in each cycle, interval $I_{0,0}$ requires three checks at $x = 12R$, i.e. one check with interval $I_{12,0}$ of P_0 , one check with interval $I_{12,0}$ of P_1 , and one check with interval $I_{0,0}$ of P_4 . In total 36 checks are required for placing P_5 . The placement of P_6 is similar to the placement of P_5 , however, in each cycle, interval $I_{0,0}$ requires 5 checks at $x = 12R$, 30 extra checks are also required to search for the indexes of the filled intervals. In total 70 checks are required for placing P_6 . The placement of P_7 is similar to the placement of P_6 , however, in each cycle, interval $I_{0,0}$ requires 7 checks at $x = 12R$, 60 extra checks are also required to search for the indexes of the filled intervals. In total 104 checks are required for placing P_7 . Table 1 presents the number of checks for placement of each piece.

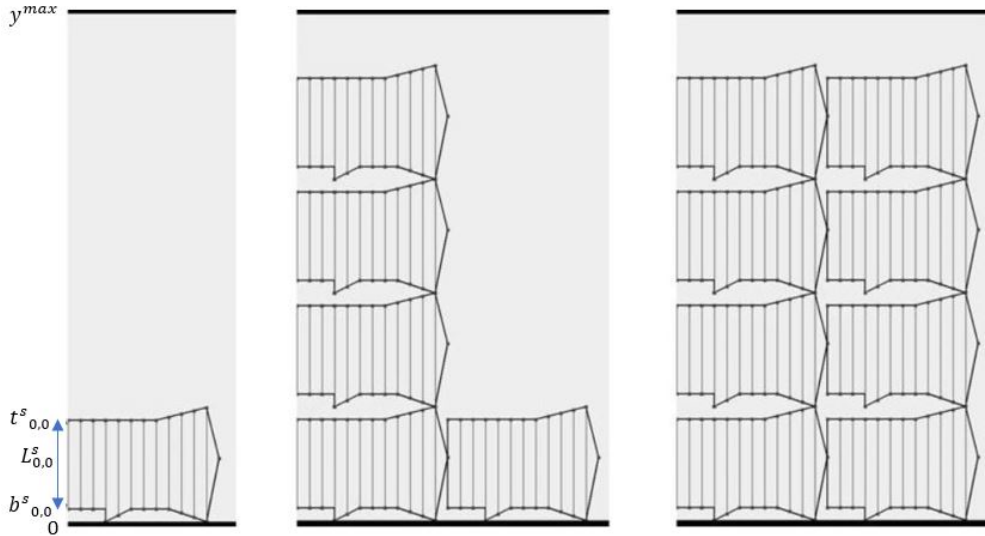


Figure 10: Placement of 8 copies of a piece. Placement of each piece starts from the translation vector of the previous piece. To check placement of interval $(b_{i,j}, t_{i,j}, P_{i,j})$ with length $L_{i,j}$, interval $(y_{max} - L_{i,j}, y_{max}]$ on each resolution line of the strip is not checked.

In case rotation of pieces is allowed, the bottom-left-fill algorithm considers all allowed rotation angles for the placement of each piece. Each piece is placed with the rotation angle that minimizes the largest strip x -coordinate of the piece. In case several rotation angles lead to the minimum, the rotation angle that minimizes the maximum strip y -coordinate of the piece is selected. In case several rotation angles satisfy these two criteria, the smallest rotation angle is selected. The decision about the optimal placement of a piece with several rotation angles is made locally, hence allowing (many) rotations does not necessarily lead to a better global solution.

Table 2

Information on the data sets used for computational experiments shown in Tables. 3,4,5.

Data set	Number of pieces	Number of different pieces	Data type of coordinates	Interval of x -projection pieces	Allowed rotation	Width of the strip
Shirts	99	8	int	[3,12]	0,180	40
Trousers	64	17	int	[6,59]	0,180	79
Swim	48	10	float	[359,1940]	0,180	5752
Han	25	20	int	[3,19]	free	58
Jakob2	25	25	int	[6,12]	free	70
poly5b	75	75	int	[3,12]	free	40
550-random	550	550	float	[1,27.2]	free	80

5. Results

In this section we discuss the performance of the proposed bottom-left-fill algorithm using semi-discrete representation, implemented in C++ using the Qt environment. Experiments were carried out on a single core of an Intel Core i7-7500U processor at 2.70 GHz and 16 GB of RAM. The computational experiments are done with the following ESICUP data sets: ‘Shirts’ Dowsland, Dowsland and Bennell (1998), ‘Han’ Han and Na (1996), ‘Trousers’ Oliveira, Gomes and Ferreira (2000), ‘Jakob2’ Jakobs (1996), ‘poly5b’ Hopper (2000) and ‘Swim’ Oliveira et al. (2000). These data sets contain both non-convex and convex pieces, vary in size and in the allowed rotation angles.

We also used a data set with 550 randomly generated pieces, these pieces are generated with 4 different diameters and 80 % of the pieces have the smallest diameter. Most pieces are non-convex. Since the coordinates of the vertices are arbitrary floating point numbers, many intervals of the semi-discretized pieces are extended using the extension algorithm, described in section 3.

Table 2 represents the information on the data sets. For the data sets with free rotation, we consider four different cases: no rotation, two rotations with $\Delta\theta = 180^\circ$, four rotations with $\Delta\theta = 90^\circ$ and eight rotations with $\Delta\theta = 45^\circ$. The base resolution R_b is computed for each rotation angle because rotation can change P_e (the smallest x -projection of all non-vertical edges of all pieces). We have used several R -values, typically up to $R_b/10$. For data sets with integer vertex coordinates, the R -values are selected such that no extension occurs if the rotation angle is a multiple of 90° . For placement the pieces are sorted according to decreasing axis-aligned bounding box area, this allows to place small pieces in the gaps between already placed large pieces.

5.1. Results for the benchmark data sets

Tables 3 and 4 show the results obtained for the five data sets: for each data set, ‘length’ and ‘time’ denote respectively the length of the strip and the execution time averaged over 100 runs. We present the time to compute the discretization and the placement time separately.

One expects that decreasing R leads to a smaller strip length. The results in Tables 3 and 4 show that this is indeed most often the case, but not always, as explained in section 5.2. On the other hand, decreasing R causes a higher computational cost. First, the computational cost of the discretization of the pieces for each rotation angle is inversely proportional to R , but the cost of the extension algorithm does not depend on R but on the number of vertices. The timings in Tables 3 and 4 shows that the execution time of the discretization is less than linear in R^{-1} . We assume that this is also partly due to a better cache hit ratio when R decreases since more tuples are computed and stored in the vector data structure holding the piece (see section 3). Second, the number of tests performed in the placement algorithm can increase more than linearly in R^{-1} , since the number of intervals to be placed increases linearly with R^{-1} , but the number of tested positions in the strip also increases with decreasing R . Table 3 (column ‘check tuples’) indicates that the actual number of tests increases slightly more than linear in R^{-1} . However, the timing in Tables 3 and 4 show that the actual execution times increase less than linear in R^{-1} due to caching effects. The placement time for n rotation angles ($n = 1, 2, 4, 8$) increases approximately linear in n .

5.2. Effect of Resolution

For a detailed study of the effect of R on the solution quality and the run-time of the algorithm, we used the ‘random’ data set, consisting of 550 pieces, see Table 2, without allowing rotation of the pieces. Note that with decreasing R

Table 3

Performance results for the data sets with two rotation angles (0° , 180°): length: length of the strip; time: time to compute semi-discretization for all rotation angles + time of the placement; check tuples: the number of checks for placing all pieces in the strip.

Data Set	Resolution R	Without Rotation			With Rotation	
		$R_b = 1$			$\Delta\theta = 180^\circ$	
		length	time(ms)	check tuples	length	time(ms)
Shirts (99 pieces)	1	70.0	0.1+0.3	13551	66.0	0.2+0.6
	0.5	69.5	0.2+0.5	25763	67.5	0.3+0.9
	0.2	68.4	0.3+0.9	72849	67.0	0.5+1.6
	0.1	68.3	0.5+1.9	156159	66.5	0.7+2.8
Trousers (64 pieces)	1	$R_b = 1$			$R_b = 1$	
		length	time(ms)	check tuples	length	time(ms)
	0.5	284.0	0.2+0.4	23127	284.0	0.3+0.8
	0.2	284.0	0.3+0.7	47065	284.0	0.5+1.4
	0.1	283.6	0.6+1.7	120623	283.6	1.1+2.8
Swim (48 pieces)	36.0	$R_b = 36$			$R_b = 36$	
		length	time(ms)	check tuples	length	time(ms)
	18.0	7687.4	0.3+0.9	78738	7255.4	0.5 + 1.7
	7.2	7597.4	0.5+1.3	119169	7417.4	0.7 + 2.4
	3.6	7651.7	0.9+3.2	294175	7478.6	1.1 + 5.2
		7565.0	1.4+6.1	610427	7496.6	1.8 +10.0

the size of the extensions decrease but also the number of places where a piece can be placed increases. The results presented in Table 5 show that the solution quality, measured by the length of the strip and the wasted space fraction W_f defined as $W_f = 100 \times \frac{\text{Area}(strip) - \sum_{i=0}^{N-1} \text{Area}(piece(i))}{\text{Area}(strip)}$, is low when R is larger than R_b , while the quality improves slowly with R^{-1} for $R < R_b$. This shows that R_b is a good starting point for selecting an appropriate R , and has diminishing returns beyond this point. The run-time of the algorithm increases less than linear in R^{-1} ; when R decreases from 1 to 0.1 the run-time increases with a factor 3.2; when R decreases from 0.1 to 0.01 the run-time increases with a factor 6.4.

When using a simple bottom-left-fill algorithm, giving priority to ‘left’ rather than ‘bottom’, decreasing R does not always lead to a smaller strip length, as shown in Fig. 11, where 8 copies of a piece are placed, after discretization with a large and a small R .

5.3. Extension

The extension of the intervals of the pieces, required to avoid the overlap of the original pieces, computed by the extension algorithm (see section 3.2) implicitly defines extension areas outside the original piece, that cannot be occupied by other (extended) pieces. To study the effect of R on these extension areas we use the ‘poly5b’ data set. We now choose R such that many convex vertices (with integer coordinates) lie between resolution lines, causing extensions in the semi-discretized representation of the pieces. The extension area decreases with decreasing R as indicated in Section 3 and illustrated in Table 6. However, the extension area is small compared to the area of the gaps between the pieces, called ‘wasted area’ in Table 6, where we also show the ‘strip area’, i.e. the length of the strip multiplied with the width of the strip.

5.4. Scalability of the algorithm

In order to test the scalability of our algorithm, in particular when the data set contains many identical pieces, we used data sets consisting of respectively 100, 200 and 400 identical pieces, the same piece as used in Fig. 11 with $R = 1$. In a first test we used the algorithm with the optimization mentioned in section 4, namely when two identical pieces are placed consecutively, the placement of the second piece starts with the translation vector of the placement of the first piece. Doubling the size of the data set, the run-time increases only slightly. In a second test we removed

Table 4

Performance results for the data sets with free rotation angles: length: length of the strip; time: time to compute semi-discretization for all rotation angles + time of the placement.

Data set	Resolution R	Without Rotation		With Rotation					
				$\Delta\theta = 180^\circ$		$\Delta\theta = 90^\circ$		$\Delta\theta = 45^\circ$	
		$R_b = 1$		$R_b = 1$		$R_b = 1$		$R_b = 0.5$	
		length	time(ms)	length	time(ms)	length	time(ms)	length	time(ms)
Han (25 pieces)	1	51.0	0.1+0.1	49.0	0.2+0.2	45.0	0.4+0.3		
	0.5	52.0	0.2+0.2	48.5	0.3+0.3	45.0	0.6+0.6	46.5	1.4+1.1
	0.2	52.0	0.3+0.4	48.2	0.6+0.7	45.2	1.2+1.3	47.0	3.0+2.6
	0.1	52.0	0.6+0.7	49.9	1.1+1.1	45.2	2.3+2.0	44.5	5.4+4.2
Jakob2 (25 pieces)		$R_b = 2$		$R_b = 2$		$R_b = 2$		$R_b = 0.5$	
		length	time(ms)	length	time(ms)	length	time(ms)	length	time(ms)
	2	30.0	0.1+0.1	30.0	0.1+0.1	30.0	0.2+0.2		
	1	28.0	0.1+0.1	28.0	0.2+0.2	31.0	0.3+0.3		
	0.5	30.5	0.2+0.3	29.0	0.3+0.4	28.0	0.4+0.6	28.0	1.1+1.0
	0.2	30.2	0.4+0.5	29.2	0.7+0.7	29.2	1.1+1.2	28.4	2.6+2.1
0.1	30.1	0.7+0.7	29.1	1.3+1.1	29.1	2.3+2.2	29.1	4.6+4.2	
poly5b (75 pieces)		$R_b = 1$		$R_b = 1$		$R_b = 1$		$R_b = 0.5$	
		length	time(ms)	length	time(ms)	length	time(ms)	length	time(ms)
	1	73.0	0.3+0.6	70.0	0.5+0.9	68.0	1.0+ 2.7		
	0.5	69.5	0.6+1.0	67.0	0.9+1.7	66.5	1.5+ 3.1	67.0	3.1+ 7.2
	0.2	70.6	0.8+2.1	69.4	1.5+4.0	65.8	2.8+ 7.4	66.0	6.3+15.5
0.1	72.4	1.5+4.1	68.5	2.9+7.4	66.0	5.3+13.4	65.9	11.3+28.5	

Table 5

Placement of 'random' data set (without rotation). The effect of R on the execution time and the solution quality: length: length of the strip; wasted space: the percentage of the wasted space in the strip; time: time to compute semi-discretization for all pieces + time of the placement.

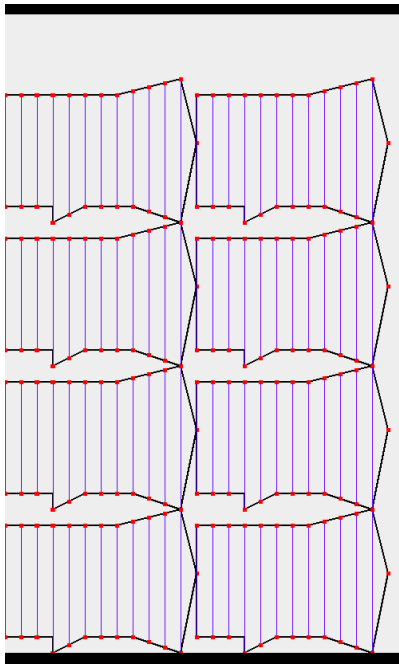
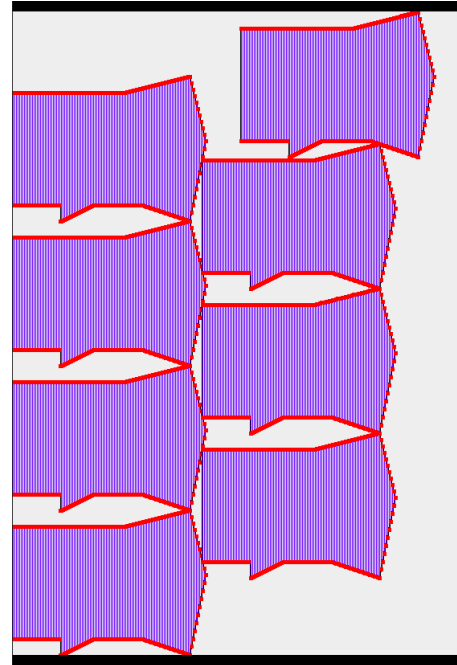
Resolution	length	wasted space W_f (%)	time (ms)
1	127.01	23.3	30
0.5	119.83	18.8	42
$R_b = 0.2$	114.83	15.2	66
0.1	112.63	13.6	98
0.05	111.93	13.0	166
0.02	111.23	12.5	350
0.01	110.74	12.1	631

Table 6

The effect of R on extension for the 'poly5b' data set (without rotation). Extension area vs. wasted area and strip area.

Resolution	Strip area	Wasted area	Extension area
1.2	3176	1340	173
0.6	2944	1108	33
0.2	2824	988	10
0.1	2896	1060	4

the above mentioned optimization, such that the placement of each piece always starts at the bottom-left corner of the strip. The resulting run-time increases superlinear, but less than quadratic, in the number of pieces; when the number of pieces increases from 100 to 200, the run-time increases with a factor 2.3 and when the number of pieces increases from 200 to 400, the run-time increases with a factor 2.5.

(a) $R = 1$, Used length of strip = 24

(b) Resolution = 0.1, Used length of strip = 26.2

Figure 11: Placement of eight copies of a piece. Using a smaller R does not always lead to a better solution.**Table 7**

Scalability of the algorithm using data sets with 100, 200 and 400 copies of a piece. Execution times for placement with and without the optimization in case of identical pieces, described in section 4.

Placement of each piece with translation vector of previous piece		Placement of each piece from the bottom-left corner of the strip	
Data set	Time (ms)	Data set	Time (ms)
100 copies	0.5	P100	1.4
200 copies	0.7	P200	3.2
400 copies	0.9	P400	8.1

5.5. Comparison with other approaches

Only few papers use a semi-discrete representation, e.g. Ma and Liu (2007); Akunuru and Babu (2013); Leao et al. (2015). We compare our results with theirs in a qualitative way. Ma and Liu (2007) and Akunuru and Babu (2013) do not give detailed information on the data sets. Although Leao et al. (2015) used a semi-discrete representation, a MIP model was used for placement of data sets with a few different pieces, which is completely different from our placement algorithm. Even without computational experiments, we can draw the following conclusions. Ma and Liu (2007) represents the pieces and the strip by axis-aligned rectangles which leads to unnecessary extensions of the pieces, compared to our extension algorithm, and the quality of the solution obtained by their bottom-left-fill algorithm (without using a metaheuristic) will be worse than the quality of the solution obtained by our algorithm. In Akunuru and Babu (2013), the extension rules do not cover all cases which can lead to overlap of pieces. In Leao et al. (2015), even for problems with a small number of pieces most of the run-times are equal to the imposed time limit and the optimal solution is not found despite using a MIP method.

For the method presented in this paper, the solution quality, i.e. the length of the strip, presented in Tables 3, 4, 5, 6 and 7 is limited by the simple greedy bottom-left-fill placement algorithm used. Many (meta)heuristics to improve the bottom-left-fill algorithm, such as reordering of pieces, tabu search, genetic algorithm, can easily be implemented on top of our placement algorithm. The semi-discretization of the pieces plays a role in the bottom-left-fill algorithm, described in section 4, but most (meta)heuristics used in the literature are independent of the representation of the

pieces and thus can use our placement algorithm as an efficient ‘building block’. The main advantages of the method, introduced in this paper, are the low run-time and the scalability w.r.t. the number of pieces.

We now compare the performance and mainly the execution time and the scalability of our algorithm with other ‘building blocks’ used in heuristics to solve nesting problems, e.g. based on the use of no-fit polygons.

In Burke et al. (2006) and Burke, Hellier, Kendall and Whitwell (2010) bottom-left-fill algorithms based on respectively arc geometry and no-fit polygon are used as ‘building block’ within a hill climbing and tabu search metaheuristic. In both papers, results are presented for the data sets mentioned above and the execution times of a single run of the bottom-left-fill algorithm, obtained on a 2 GHz Intel Pentium 4 processor with 256 MB RAM, are given, but not the strip length achieved in such a single run.

In Burke et al. (2006) the positions to place a piece in the strip are limited to a set of equidistant vertical lines, as in our algorithm. The placement of each piece starts from the bottom-left corner of the strip. If the piece intersects with already placed pieces, intersections of the geometric primitives are resolved by shifting the piece upward and, if necessary to the right on the next resolution line, until a valid position is found, which is equivalent with our bottom-left-fill implementation. In Burke et al. (2010) no-fit polygon is used to identify the intersection state of two pieces. However, the time for computing the NFP of pieces which is discussed in Burke, Hellier, Kendall and Whitwell (2007) is not taken into account in the timings mentioned in Burke et al. (2010). For several ESICUP data sets, Table 8 shows the run-time of a single run of the bottom-left-fill placement presented in Burke et al. (2006) and Burke et al. (2010), the best result obtained by using hill climbing and tabu search on top of the bottom-left-fill algorithm, and the corresponding run-time reported in Burke et al. (2006), and the best result obtained with our algorithm, cf. Tables 3 and 4, and the corresponding run-time.

When comparing the quality obtained with our algorithm and with a bottom-left-fill algorithm using continuous representation, one must make distinction between the effect of the discretization of the pieces and the discretization of the strip. When semi-discretization of a piece causes extensions, no other piece can be placed in the extension regions, but we indicated in section 5.3 that the area of the extension regions typically is small, and for pieces with integer vertex coordinates, as in most of the ESICUP data sets, R can be chosen such that no extension occurs. Semi-discretization of the strip limits the possible placement positions, but the results in Tables 4 and 5 show that, for sufficiently small R , decreasing R , i.e. increasing the number of possible placement positions, does not always lead to a shorter strip length, due to the greedy nature of the bottom-left-fill heuristic.

In Burke et al. (2006) the strip length achieved with a single run of the bottom-left-fill algorithm is not presented, but this length should be equal to the length obtained with our bottom-left-fill algorithm, when in both Burke et al. (2006) and our method the same distance between the vertical lines in the strip and the same ordering of the pieces are used, and if no extension occurs in the semi-discretization of the pieces in our algorithm. The latter is true for the results in Table 9 for the data sets Shirts, Trousers, Jakob2 and Poly5b.

Comparison of the execution times of both bottom-left-fill algorithms is not straightforward: R and the ordering of the pieces are not mentioned in Burke et al. (2006) and the performance ratio of the processors used is not known, since this depends on many factors (data structures used, compiler options, cache sizes, ...). The performance ratio of the processor used for our experiments and the one used in Burke et al. (2006) and Burke et al. (2010) is not larger than 25, see the detailed analysis in Appendix. Taking this performance ratio into account, the timings for a single bottom-left-fill placement indicate that our placement algorithm using semi-discretization is two orders of magnitude faster than the bottom-left-fill placement algorithm in Burke et al. (2006). Considering the timings for computing the no-fit polygon of pieces discussed in Burke et al. (2007), our placement algorithm using semi-discretization is at least an order of magnitude faster than the bottom-left-fill algorithm in Burke et al. (2010). The efficiency of our algorithm is mainly due to the limited number of tests needed on average to evaluate the placement of a piece in a certain position in the strip and the use of simple operations on data structures that allow to achieve high performance on current computers (for example, ensuring a high cache hit ratio).

We also compare our algorithm with other recently proposed ‘building blocks’. Cherri, Carravilla and Toledo (2016) combine exact and heuristic approaches using a dotted board model. Their method has three phases: constructive phase, improvement phase and compaction phase. The constructive phase computes a feasible solution and can be considered as a building block. The only common data sets used in Cherri et al. (2016) and this paper are "Jakob2" and "Trousers". The data in Table 8 indicate that our algorithm is orders of magnitude faster than the ‘building block’ in Cherri et al. (2016) and results in a shorter strip length.

For completeness, we present in Table 8 also results reported in recent papers, in which not only the solution quality but also the execution time of the placement are presented. Although these papers use much more powerful algorithms

Table 8

Comparison of the best results from Tables 3 and 4 with the best results obtained in the literature: run-time: time in ms; BL: Bottom-Left.

		Data sets				
		Shirts	Trousers	Swim	Jakob2	poly5b
Best results from Tables 3 and 4	length	66.0	283.6	7255.4	28.0	65.8
	run-time	0.8	2.3	2.2	0.2	10.2
Burke et al. (2006) (BL only)	length	NA	NA	NA	NA	NA
	run-time	4990	7890	12390	2130	14700
Burke et al. (2006) (BL + metaheuristics)	length	63.8	246.6	6462.40	25.8	60.5
	run-time	58360	756150	607370	81410	676610
Burke et al. (2007, 2010) (BL only)	length	NA	NA	NA	NA	NA
	run-time	330+770	730+1020	6080+1240	5070+640	141900+12620
Best results from Cherri et al. (2016)	length	NA	439.0	NA	36.0	NA
	run-time	NA	229900	NA	34100	NA
Best results from Sato et al. (2016a)	length	65.0	253.1	6772.4	24.1	NA
	run-time	21600000	21600000	21600000	7200000	NA
Best results from Sato et al. (2019)	length	60.4	239.2	5846.8	22.0	NA
	run-time	1200000	1200000	1200000	600000	NA
Best results from Kierkosz and Luczak (2019)	length	55.2	232.7	5474.9	22.2	NA
	run-time	37930	408350	530500	192530	NA

than bottom-left-fill, we include these results to show the best solution quality presented in the literature for the data sets that we use.

Sato, Martins and Tsuzuki (2016a) uses a pairwise placement strategy in which one item is always positioned in exact fitting or sliding placements, which are positions where the item movement is restricted. The no-fit polygon (NFP) is employed in combination with the inner-fit rectangle (IFR) in order to guarantee the validity of the layout. A simulated annealing algorithm controls the placement sequence and guides the search over the solution space.

In Sato et al. (2019), a separation and compaction strategy is used to solve the strip packing problem. This strategy employs a raster penetration map to efficiently determine the overlap. The penetration depth is computed using the discrete nofit polygon and an overlap function which is a collision penalty function defined for a pair of items. The common data sets used in this paper, Sato et al. (2016a) and Sato et al. (2019) are "Jakob2", "Shirts", "Swim" and "Trousers".

Kierkosz and Luczak (2019) propose a one-pass algorithm. The no-fit polygon of the partially packed strip and a new piece is computed. The new piece is placed at the vertex of the no-fit polygon for which a 'fitting function' is maximized. The common data sets used in Kierkosz and Luczak (2019) and this paper are "Jakob2", "Shirts", "Swim" and "Trousers".

6. Conclusion

We have shown that a semi-discretization of both the pieces and the strip can lead to a fast and scalable method to solve the 2D nesting problem. First, we presented a fast algorithm to compute the semi-discretization of possibly non-convex pieces, consisting of equidistant vertical intervals, complemented with an extension procedure, ensuring that a non-overlapping placement of the intervals guarantees that the original pieces do not overlap. The proposed extensions are minimal and complete, which is an improvement over extension techniques already proposed in the literature. Note that changing the shape of the strip from rectangular irregular, as discussed in Baldacci, Boschetti, Ganovelli and Maniezzo (2014), is easy in our approach, since it only requires the semi-discretization of an irregular shaped strip, without extension, which would lead to only a small amount of loss in strip space. Further, we presented an efficient bottom-left-fill placement algorithm in detail. It exploits the fact that only simple arithmetic operations are needed to detect and avoid overlap and it uses an optimised ordering of the interval overlap tests. Much attention is paid to the choice of the data structures in order to achieve high performance on current processors. The quality of the solution and the execution time of the algorithm greatly depend on R , i.e. the distance between the vertical intervals.

However, the execution time increases less than linear in R^{-1} . We also show how a suitable ‘base resolution’ can be computed from the size and geometry of the pieces. Although, the quality of the solutions is limited by the simple greedy algorithm used, timings of the bottom-left-fill placement for several ESICUP data sets show that our placement algorithm using a semi-discrete representation is nearly two orders of magnitude more efficient than a similar algorithm based on arc geometry, requiring the computation of intersections of the original pieces. Moreover, since the execution times scale well with increasing number of pieces, our placement algorithm is scalable. Therefore, due to the low run-time and the scalability of our algorithm, the placement algorithm presented in this paper can be used as an efficient ‘building block’ to implement various (meta)heuristics already proposed in the literature. We have implemented the hill climbing heuristic, used in Burke et al. (2006) and Burke et al. (2010), on top of our bottom-left-fill algorithm and initial experiments show that we are able to obtain similar quality solutions in a very low run-time. This proves two things; firstly, the proposed building block is fast and secondly, the proposed extensions are very minimal that do not reduce the quality of the solutions.

Acknowledgment

We acknowledge the financial support of Interne Fondsen KU Leuven / Internal Funds KU Leuven, project C24/17/048.

References

- Akunuru, R., Babu, N., 2013. Semi-discrete geometric representation for nesting problems. *International Journal of Production Research* 51, 4155–4174.
- Amaro Junior, B., Pinheiro, P.R., Saraiva, R.D., Pinheiro, P.G.C.D., 2014. Dealing with nonregular shapes packing. *Mathematical Problems in Engineering* Volume 2014, Article 548957. doi:<https://doi.org/10.1155/2014/548957>.
- Baldacci, R., Boschetti, M., Ganovelli, M., Maniezzo, V., 2014. Algorithms for nesting with defects. *Discrete Applied Mathematics* 163, 17–33.
- Bennell, J., Oliveira, J., 2008. The geometry of nesting problem: a tutorial. *European Journal of Operational Research* 184, 397–415.
- de Berg, M., Cheong, O., van Kreveld, M., Overmars, M., 2008. *Computational Geometry: Algorithms and Applications*. Springer. Chapter 2.
- Burke, E., Hellier, R., Kendall, G., Whitwell, G., 2006. A new bottom-left-fill heuristic algorithm for the two-dimensional irregular packing problem. *Operations Research* 54, 587–601.
- Burke, E., Hellier, R., Kendall, G., Whitwell, G., 2007. Complete and robust no-fit polygon generation for the irregular stock cutting problem. *Operations Research* 179, 27–49.
- Burke, E., Hellier, R., Kendall, G., Whitwell, G., 2010. Irregular packing using the line and arc no-fit polygon. *Operations Research* 58, 948–970.
- Cherri, L., Carravilla, M., Toledo, F., 2016. A model-based heuristic for the irregular strip packing problem. *Pesquisa Operacional* 36, 447–468.
- Cplusplus.com, 2020. class template std::vector. <http://www.cplusplus.com/reference/vector/vector/>. Last accessed 22 November 2020.
- Dowland, K., Dowland, W., Bennell, J., 1998. Jostling for position: Local improvement for irregular cutting patterns. *The Journal of the Operational Research Society* 49, 647–658.
- Gennady, F., Shaojuan, Z., Abhinav, S., 2016. Intel Math Kernel Library (Intel MKL) benchmarks suite. <https://software.intel.com/content/www/us/en/develop/articles/intel-mkl-benchmarks-suite.html>. Last accessed 22 November 2020.
- Gomes, A.M., 2013. Irregular packing problems: Industrial applications and new directions using computational geometry. *IFAC-Proceedings* 46, 378–383.
- Gomes, A.M., Oliveira, J., 2002. A 2-exchange heuristic for nesting problems. *European Journal of Operational Research* 141, 359–370.
- Hager, G., Wellein, G., 2011. *Introduction to High Performance Computing for Scientists and Engineers*. Chapman and Hall/CRC. Chapter 1-3.
- Han, G.C., Na, S.J., 1996. Two-stage approach for nesting in two-dimensional cutting problems using neural network and simulated annealing. *Journal of Engineering Manufacture* 210, 509–519.
- Harper, D., 2006. Pentium floating-point speed test. <https://www.obliquity.com/computer/speedtest.html>. Last accessed 22 November 2020.
- Hopper, E., 2000. Two-dimensional packing utilising evolutionary algorithms and other meta-heuristic methods. Ph.D. thesis. University of Wales, Cardiff School of Engineering.
- Jakobs, S., 1996. Genetic algorithms for the packing of polygons. *European Journal of Operational Research* 88, 165 – 181.
- Kierkosz, I., Luczak, M., 2019. A one-pass nesting problems. *Operations Research And Decisions* 29, 37–60.
- Leao, A., Toledo, F., Oliveira, J., Carravilla, M., 2015. A semi-continuous MIP model for the irregular strip packing problem. *International Journal of Production Research* 54, 712–721.
- Leao, A., Toledo, F., Oliveira, J., Carravilla, M., Alvarez-Valdés, R., 2020. Irregular packing problems: A review of mathematical models. *European Journal of Operational Research* 282, 803–822.
- Ma, H., Liu, C., 2007. Fast nesting of 2-D sheet parts with arbitrary shapes using a greedy method and semi-discrete representations. *IEEE Transactions on Automation Science And Engineering* 4, 273–282.
- Oliveira, J.F., Gomes, A.M., Ferreira, J.S., 2000. A new constructive algorithm for nesting problems. *OR Spektrum* 22, 263–284.
- Pinheiro, P.R., Amaro Junior, B., Saraiva, R.D., 2016. A random-key genetic algorithm for solving the nesting problem. *International Journal of Computer Integrated Manufacturing* 29, 1159–1165.

Table 9

Sweep-line algorithm: different cases for the update of the ActiveEdges data structure

Case	Position of vertex, associated with Events [m], on its adjacent edges	Update of ActiveEdges
a	Events [m] is the x -coordinate of the left endpoint of two edges, e_1 and e_2 .	Add information of edges e_1 and e_2 .
b	Events [m] is the x -coordinate of the right endpoint of edge e_1 and the left endpoint of edge e_2 .	Remove information of edge e_1 and add information of edge e_2 .
c	Events [m] is the x -coordinate of the right endpoint of two edges, e_1 and e_2 .	Remove information of edges e_1 and e_2 .
d	Events [m] is the x -coordinate of the left endpoint of edge e_2 and edge e_1 is vertical.	Add information of edge e_2 .
e	Events [m] is the x -coordinate of the right endpoint of the edge e_1 and edge e_2 is vertical.	Remove information of edge e_1 .

Sato, A., Martins, T., Gomes, A., Tsuzuki, M., 2019. Raster penetration map applied to the irregular packing problem. *European Journal of Operational Research*. 279, 657–671.

Sato, A., Martins, T., Tsuzuki, M., 2016a. A pairwise exact placement algorithm for the irregular nesting problem. *International Journal of Computer Integrated Manufacturing*. 29, 1177–1189.

Sato, A.K., Tsuzuki, M.S.G., Martins, T.C., Gomes, A.M., 2016b. Study of the grid size impact on a raster based strip packing problem solution. *IFAC-PapersOnLine*. 49, 143–148.

Toledo, F., Carravilla, M., Ribeiro, C., Oliveira, J., Gomes, A.M., 2013. The dotted-board model: A new MIP model for nesting irregular shapes. *International Journal of Production Economics*. 145, 478–487.

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

Wauters, T., Uyttersprot, S., Esprit, E., 2016. JNFP: a robust and open-source Java based nofit polygon generator library. 13th ESICUP meeting https://paginas.fe.up.pt/~esicup/extern/esicup-13thMeeting/uploads/Conference/13th_ESICUP_Meeting_booklet_2016.pdf.

A. Appendix

The intersection of the sweep-line, i.e. a vertical line traversing the piece from left to right at $x_i = i \times R$, $i = 0, 1, 2, \dots$, with the edges of the piece gives the y -coordinates $(b_{i,j}, t_{i,j})$ of the bottom and top endpoints of the j -th line segment on the i -th resolution line. A vector `Events` is created containing the x -coordinates of the vertices of the polygonal piece (in case several vertices have equal x -coordinate it is added once), sorted in increasing order. The vertical sweep-line traverses the piece from left to right and the algorithm maintains a vector `ActiveEdges`, containing the information of the edges that exist for the x -coordinates in the interval $[\text{Events}[m], \text{Events}[m+1]]$, sorted from bottom to top. `ActiveEdges` must be updated when the sweep-line passes through an event.

The actions required to update `ActiveEdges` when the sweep-line passes through `Events[m]` depend on the position of the vertex, associated with `Events[m]`, w.r.t. its adjacent edges. Five cases must be distinguished, and the corresponding actions are listed in Table 9 and illustrated in Figs. 12-16. If `Events[m]` is the x -coordinate of the left endpoint of a non-vertical edge e_i , the information of edge e_i is added to `ActiveEdges`. Also, the information of the non-vertical edges which have `Events[m]` as the x -coordinate of their right endpoint will be removed from `ActiveEdges`. In case `Events[m]` is the x -coordinate of the endpoints of a vertical edge, the edge is not added to `ActiveEdges`.

After updating `ActiveEdges`, for every resolution line between the current event and the next event, the y -coordinates of the intersection points of the resolution line and the edges in `ActiveEdges` are computed and saved in a vector `Ypoints`.

Special cases occur when vertices lie on a resolution line, i.e. when an event occurs at $x_i = i \times R$, $i = 0, 1, 2, \dots$. In such cases, `Ypoints` is computed as follows. If vertex v_j is the left endpoint (see Fig. 12) or the right endpoint (see Fig. 14) of both adjacent edges, the y -coordinate of the vertex is added twice to `Ypoints` (indicating a segment of

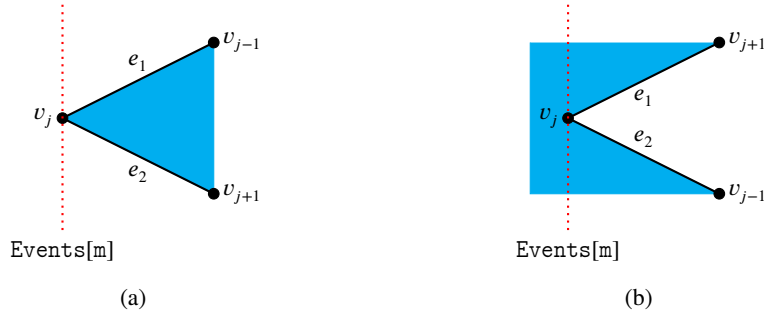


Figure 12: Updating ActiveEdges: case a: Events[m] is the x-coordinate of the left endpoint of two edges, e_1 and e_2 . Information of edges e_1 and e_2 are added to ActiveEdges.



Figure 13: Updating ActiveEdges: case b: Events[m] is the x-coordinate of the right endpoint of edge e_1 and the left endpoint of edge e_2 . Information of edge e_1 is removed and information of edge e_2 is added to ActiveEdges.

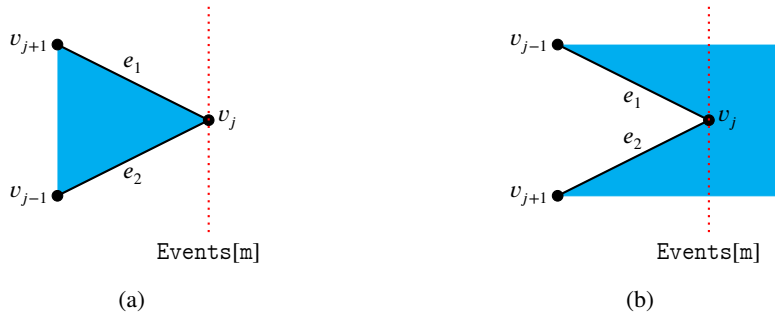


Figure 14: Updating ActiveEdges: case c: Events[m] is the x-coordinate of the right endpoint of two edges, e_1 and e_2 . Information of edges e_1 and e_2 are removed from ActiveEdges.

length zero when v_j is convex). In case a vertical edge lies on a resolution line, the y-coordinates of its endpoints are added to vector Ypoints as follows. If vertex v_j is the upper endpoint of a vertical edge and the piece occupies the space above v_j (see Fig. 15b), then the y-value of vertex v_j is added twice to Ypoints, otherwise it is added once (see Fig. 15a). If vertex v_j is the lower endpoint of a vertical edge and the piece occupies the space below v_j (see Fig. 16b), then the y-value of vertex v_j is added twice to Ypoints, otherwise it is added once (see Fig. 16a).

Traversing the intersection points in Ypoints according to increasing y-coordinate gives the y-intervals $(b_{i,j}, t_{i,j})$ representing the y-coordinates of the endpoints of the line segments of the piece, where elements with even indexes in Ypoints ($b_{i,j}$) are the lowest points of the intervals and elements with odd indexes in Ypoints are the highest points ($t_{i,j}$). We assign a label to each interval, representing the position of the interval in the semi-discretized piece. This label will be used during placement to allow that pieces touch each other, see section 4. Therefore, each interval is stored as a tuple $(b_{i,j}, t_{i,j}, P)$, with label P . Considering the position of the intervals in a piece gives three different values for P . The label of an interval on a resolution line between the current and the next event is M (Middle). On a resolution line coinciding with an event, a zero length interval created by a convex vertex v_j has the position label R

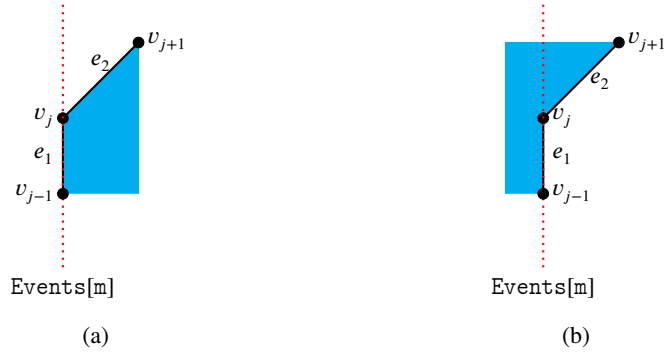


Figure 15: Updating ActiveEdges: case d: Events[m] is the x-coordinate of the left endpoint of edge e_2 and edge e_1 is vertical. Information of edge e_2 is added to ActiveEdges.

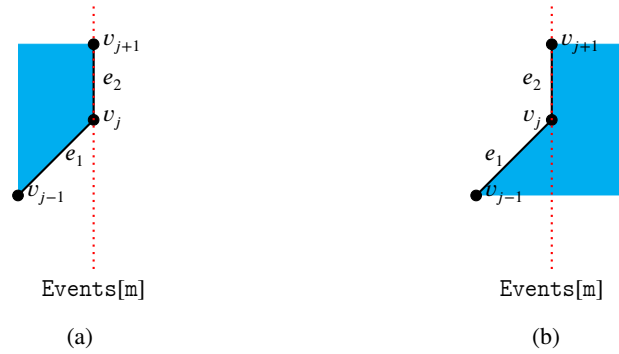


Figure 16: Updating ActiveEdges: case e: Events[m] is the x-coordinate of the right endpoint of edge e_1 and edge e_2 is vertical. Information of edge e_1 is removed from ActiveEdges.

(Right), if both adjacent edges lie to the right of v_j , has the label *L* (Left), if both adjacent edges lie to the left of v_j . An interval corresponding to a vertical edge has the label *R* (Right), if the inside of the piece lies to the right of the edge, has the label *L* (Left), if the inside of the piece lies to the left of the edge. The tuples at $x_i = i \times R$, $i = 0, 1, 2, \dots$, are saved in a vector Piece[i], which is an element of the vector Piece. Hence at the end of the sweep-line algorithm, Piece contains all vectors of tuples of the piece.

B. Appendix

In order to compare the performance of one core of the Intel Core i7-7500U processor, used for our experiments, with the Pentium 4 processor, used in Burke et al. (2006), we rely on actual performance benchmark results for the former and published results for the latter processor. A complete specification of the Intel Core i7-7500U processor can be found at <https://ark.intel.com/content/www/us/en/ark/compare.html?productIds=95451>: 2 cores, 4 threads, 2.7 GHz base frequency, 3.5 GHz max. turbo frequency, 4 MB Intel Smart Cache, 16 Gbyte memory, 34.1 GB/s max. memory bandwidth. We do not have the complete specification (cache size, memory bandwidth, ...) of the Pentium 4 (2 GHz) processor used in Burke et al. (2006)).

The number of floating-point operations achieved on the Linpack benchmark can be considered as the peak performance that a processor can achieve on an actual application code. In Gennady, Shaojuan and Abhinav (2016) performance of a Pentium 4 (2 GHz) processor on the Linpack benchmark is presented for matrix sizes up to 1804 KB. Assuming that the matrix is stored in double precision (64 bit words) the latter corresponds to a matrix of dimension 475. The highest floating point rate (0.66 Gflops = $0.66 \cdot 10^9$ floating points operations per second) is achieved for matrix dimension ≈ 155 . For matrices of dimension 100 the floating point rate is ≈ 0.55 Gflops, while for matrices larger than 280 the floating point rate is ≈ 0.25 Gflops.

On the Intel Core i7-7500U processor with 2 cores (4 threads) we have used the optimized Linpack benchmark code downloaded from <https://software.intel.com/content/www/us/en/develop/articles/intel-mkl-benchmarks-suite.html>. For matrices of dimension 100 we measured a floating point rate of ≈ 10 GFlops. For larger matrix dimensions the floating point rate increases, e.g. for matrix dimension 10000 we measured a floating point rate of ≈ 88 GFlops. Hence, for small matrix sizes, the floating point performance of the Intel Core i7-7500U processor with 2 cores is approximately a factor 20 higher than of a 2 GHz Pentium 4 processor. However, since we performed the computational experiments on one core of this processor we may assume that the floating point performance on one core is approximately a factor 10 higher than the performance of a 2 GHz Pentium 4 processor.

Performance on the Linpack benchmark is not representative for the performance on the placement procedures, since in the latter case only few operations are performed per data element fetched from memory. Therefore we also compare the performance of both processors on a C code, designed to run under Linux, performing simple operations on elements of two arrays, downloaded from Harper (2006), where also the performance obtained on a 2.8 GHz Pentium 4 processor is given. On the latter processor, performing 100 times the addition of two arrays of length 10^7 requires 6.73 seconds, which corresponds to 148 Mflops. Based on this we estimate the floating point rate of a 2 GHz Pentium processor for this operation at 106 Mflops.

On a single core of the Intel Core i7-7500U processor the same operation requires 1.14 seconds, which corresponds to 876 Mflops (code compiled using gcc with optimisation level 'O3'). Comparison of the performance of also other operations on elements of two arrays on both processors, indicates that the floating point performance ratio of a single core of the Intel Core i7-7500U processor and a 2 GHz Pentium 4 processor is approximately a factor 8.

Since these performance comparisons may give an incomplete picture of the performance difference between both processors for the implementation of the placement algorithms, we have used in section 4 the factor 25 for the performance ratio, which is probably an over-estimation.

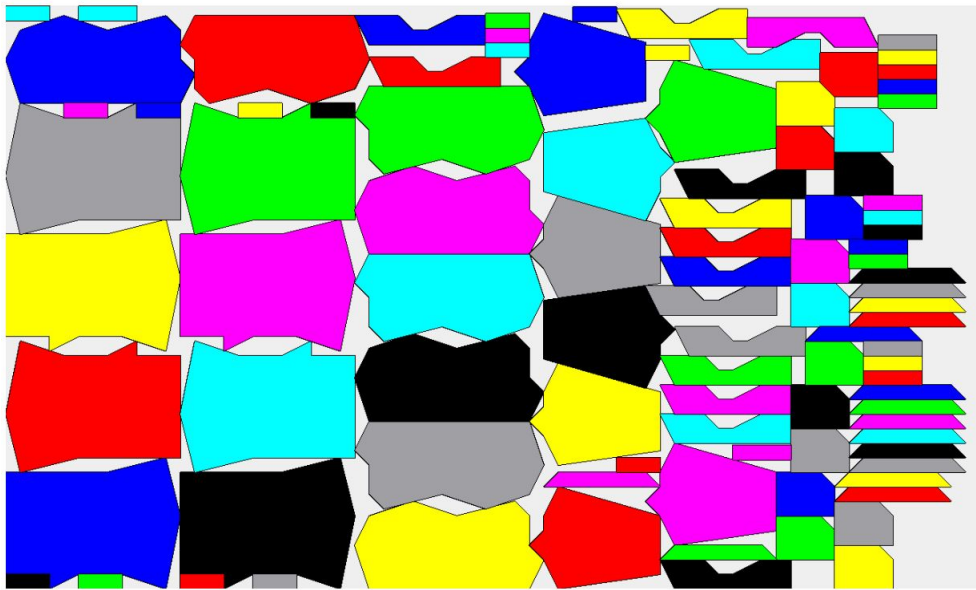


Figure 17: Placement result for dataset 'Shirts' semi-discretized with $R = 1$ and 2 allowed rotations ($\Delta\theta = 180^\circ$). Resulting strip length = 66.



Figure 18: Placement result for data set 'Trousers' semi-discretized with $R = 0.2$ and 2 allowed rotations ($\Delta\theta = 180^\circ$). Resulting strip length = 283.6.

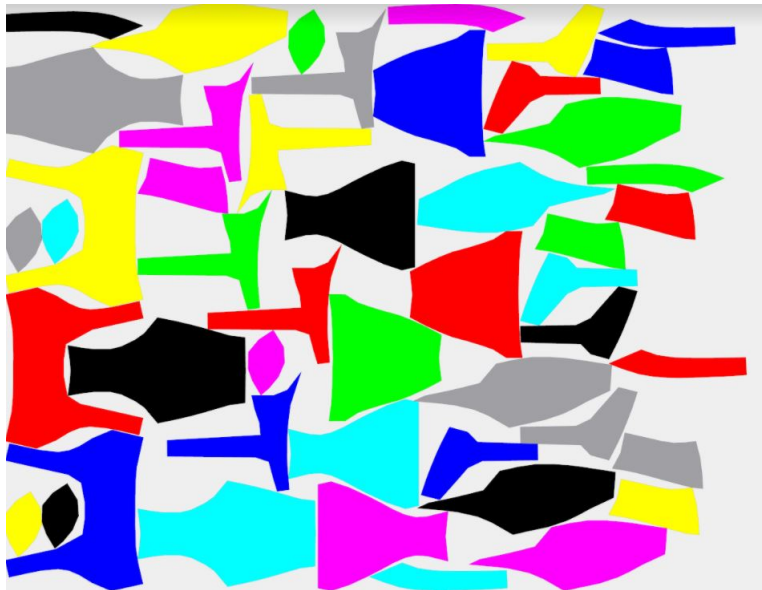


Figure 19: Placement result for data set 'Swim' semi-discretized with $R = 36$ and 2 allowed rotations ($\Delta\theta = 180^\circ$). Resulting strip length = 7255.4.

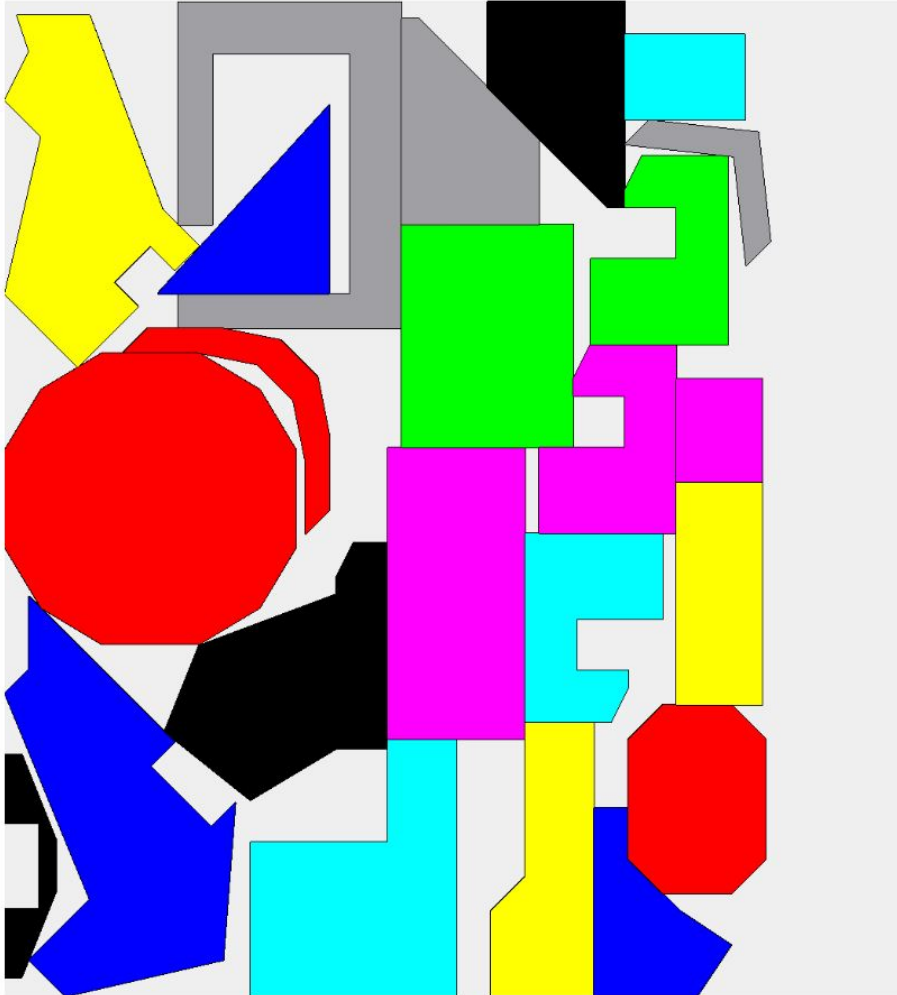


Figure 20: Placement result for data set 'Han' semi-discretized with $R = 0.1$ and 8 allowed rotations ($\Delta\theta = 45^\circ$). Resulting strip length = 44.5.

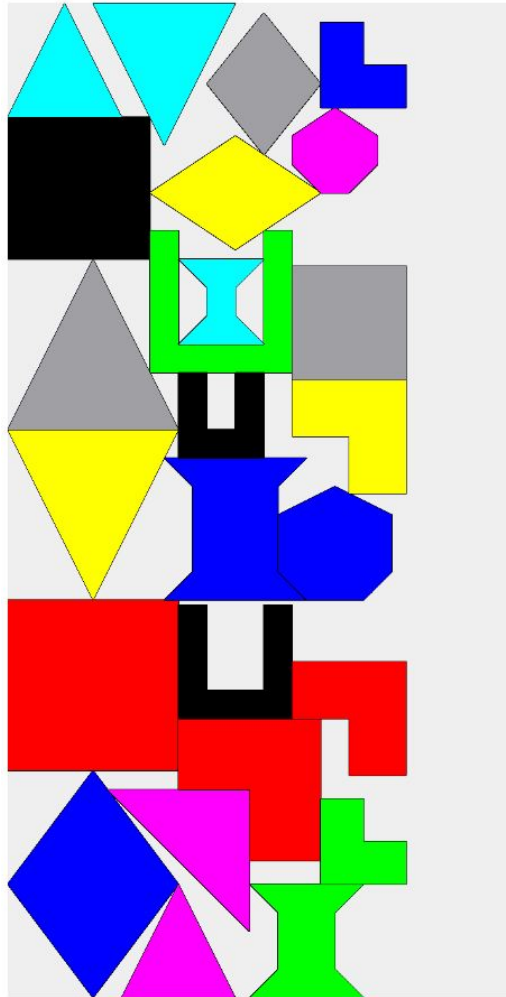


Figure 21: Placement result for data set 'Jakob2' semi-discretized with $R = 1$ and no rotation allowed. Resulting strip length = 28.0.

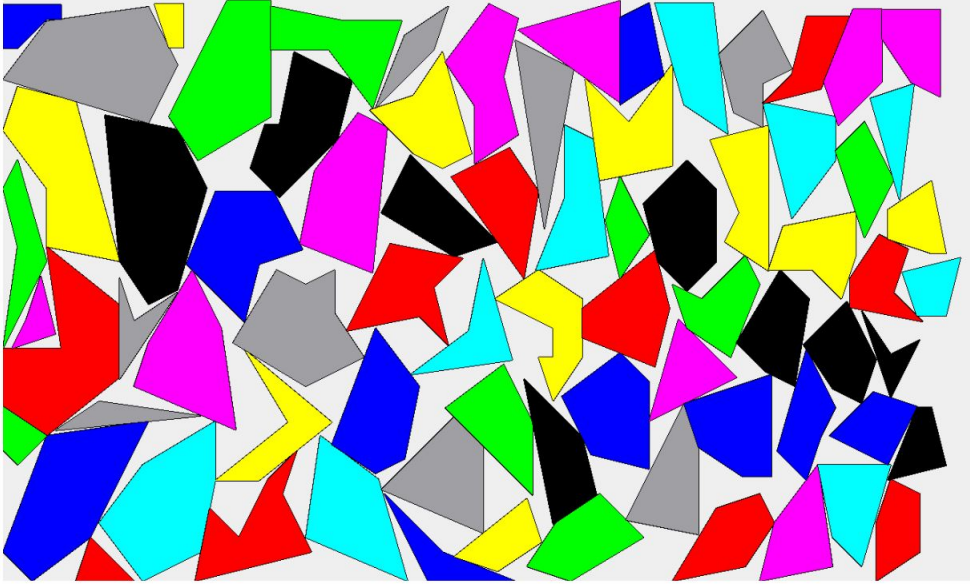


Figure 22: Placement result for data set 'poly5b' semi-discretized with $R = 0.2$ and 4 allowed rotations ($\Delta\theta = 90^\circ$). Resulting strip length = 65.8.

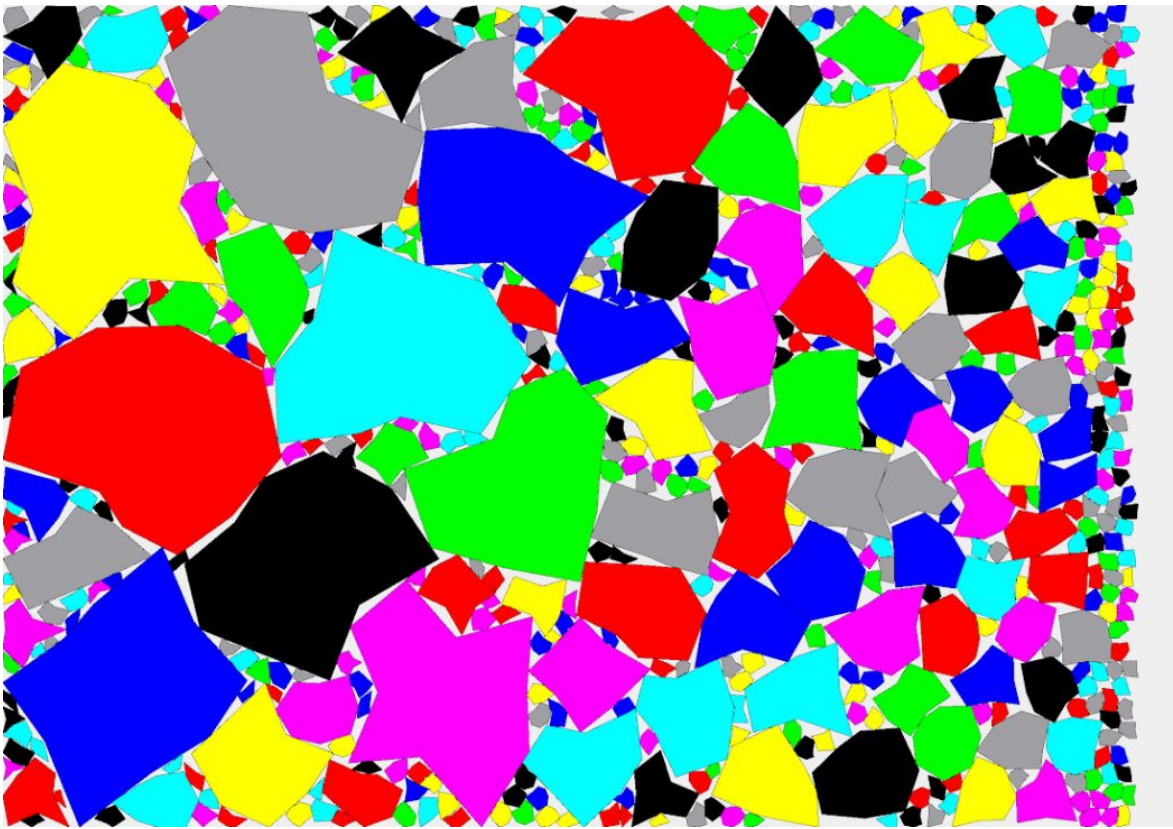


Figure 23: Placement result for random data set with 550 pieces, semi-discretized with $R = 0.01$ without rotations allowed. Resulting strip length = 110.74.