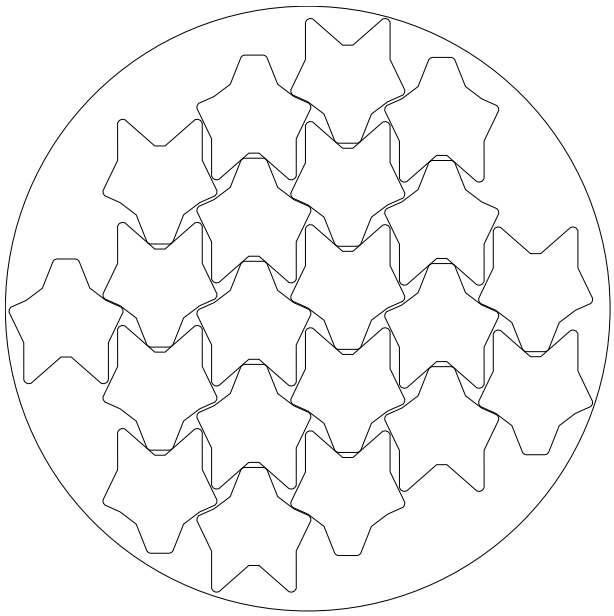


Nesting Irregular Shapes in a Circular Area

Dr. Markus Säbel

Masterarbeit

FernUniversität in Hagen
Studiengang Praktische Informatik M.Sc.
Matrikelnr. 6409016

Februar 2022

Contents

Chapter 1

Introduction

The purpose of this thesis is to examine a special variant of the nesting problem. I will first lay out the practical motivation for the problem and then contextualize it with respect to existing problem formulations and solutions.

1.1 Motivation

The practical task requiring an algorithmic solution was to optimize the output of a machine for the production of multi-axial non-crimp carbon fiber textiles developed by the Kejora GmbH in Stuttgart. The Coyotex™ technology offers several benefits over existing technologies:

- Production is near-net shape, i.e. instead of cutting preforms from continuous or rectangular stock, they are layed out individually using 22 mm fiber tape, significantly reducing waste of an expensive material.
- Fiber orientation can be chosen freely for each layer, allowing it to be calibrated towards the particular demands of a given application.
- The optical appearance of the textile is highly customizable. By stacking partial layers with different orientations, customers can create a wide variety of optical effects without affecting the performance of the material.

In order to facilitate variable fiber orientation, preforms are layed out on a circular production table.[1] One important constraint on the productivity

[1] See the title page for an illustration of a typical nesting result. The circle represents the edge of the production table. The polygons represent the offset curves of instances of the trademark coyote head preform.

of the machine is the number of preforms that can be produced simultaneously on this table, which creates a variant of the nesting problem with the following characteristic constraints:

1. The area into which preform polygons are nested is a circle with a diameter of 2640 mm.[2] This distinguishes the problem from existing formulations, which mostly deal with rectangular or continuous stock.

2. Since fiber orientation has to be maintained between preforms, they cannot be freely rotated during nesting—the only permissible rotation is by 180°.

3. Since preforms are essentially approximated in a grid of 22 mm × 11 mm pixels, they cannot be nested too closely. In practice we have been working with a clearance of 22 mm around each preform, which is equivalent to a minimum distance of 44 mm between preforms to avoid overlap. This is easy to implement when preforms are approximated by simple geometrical primitives like rectangles or circles, but for more advanced forms of nesting it requires computing an offset curve for the preform polygon.

4. Because of the high variability in the composition of layers, the problem is, for practical purposes, restricted to the nesting of identical preforms.

5. The algorithm should be fast. Since preforms can be customized and ordered from a webshop and pricing is calculated on the spot based on the nesting result, at least a rough nesting result should be available in < 1 s.

1.2 Geometric Formulation

Geometrically, the central problem can be conceptualized as follows:

1. The production table is a circle with a certain diameter.

[2] This corresponds to a configuration of 120 tracks. More recently, for technical reasons the specification was changed to 117 tracks, equalling a table diameter of 117×22 mm $=$ 2574 mm, but I have retained the older configuration for the sake of comparability with earlier results. The same goes for the specification of part clearance (point 3), which was recently changed to 33 mm. It goes without saying that both parameters can be adjusted easily in the algorithm.

2. Individual preforms are sets of polygons. One polygon represents the outline of the preform, the others, if present, represent holes in the preform. All polygons are assumed to be simple. No two polygons touch or intersect. If there is more than one polygon, one polygon contains the other polygons in its interior and no other polygon contains any other polygon.[3]

3. The task is to fit as many outline polygons into the circle representing the production table while maintaining a minimum distance between polygons of 44 mm and a distance of 22 mm to the edge of the table.

4. Preform polygons can be translated arbitrarily inside the circle. The only permissible rotation is by 180°.

Problems of this kind are discussed under different names in the literature. When the objects that are to be fitted inside a container have a regular shape, the problem is often referred to as a "packing problem". When the object has irregular shape, the term "nesting" is used.[4] There is also a close connection to the so-called "cutting stock problem", but in our case no stock is actually cut. Since preforms are often irregular in shape, I have used the term "nesting" both for the general problem and for the specific strategy utilizing an offset curve of the preform. I have used "packing" when the preform is approximated by a regular shape, namely a rectangle or circle.

1.3 Implementation

Early versions of the box and circle packing algorithms were implemented in Java and executed as a JAR file in a Node.js container. Customer data was encoded in JSON format by the JavaScript front end and fed to the JAR file as command-line argument. The nesting result was printed to the console and read back by the front end. In the next iteration, implementing the algorithms as servlets on a separate Apache web server is envisioned, but this is not part of the thesis, which focusses on the algorithms.

Since essentially an algorithm is something that computes a function, the most appropriate implementation in an object-oriented language like Java would be as a class with static-only methods. But in some cases we want the algorithm to retain more information about the nesting process than just the

[3]Since we are only nesting identical preforms, for the purposes of nesting the algorithm can ignore all polygons representing holes—no preform can be nested inside another preform.

[4]See [1], for example.

end result, for example for the purposes of logging or generating illustrations, in which case algorithms can be instantiated as objects. Although there really aren't any *bona fide* objects in computational geometry, object-orientation is useful for modeling geometric objects and bundling the methods operating on them. I have chosen to always make the basic properties of geometrical objects final, so translating or rotating a polygon always generates a new polygon from the old one. One upside of this is that there is no need for getters and setters for the basic fields; they can be declared public, which makes for a much nicer syntax for mathematical calculations.

In general, the discussion in this thesis is on the algorithmic level, i.e. I don't discuss details of the implementation. All algorithms were implemented in Java and the source code plus a full Javadoc documentation of the classes are available for anyone interested in these details.

Chapter 2

Box Packing

In box packing, the irregular shape of the preform is approximated by a minimum-area oriented rectangle (minimum bounding box) and copies of this rectangle are then fitted into the circle representing the production table. There are two subproblems:

1. calculating the minimum bounding box and
2. maximizing the number of placed boxes.

The complexity of calculating the minimum bounding box is greatly reduced by the fact that (at least) one of the sides of a polygon's minimum bounding box must be collinear with an edge of its convex hull polygon.[1] So in order to calculate the the minimum bounding box efficiently, we calculate its convex hull first.

2.1 Convex Hull

The convex hull can be defined as the smallest convex superset of a set of points. In terms of geometric figures, the convex hull is the smallest convex polygon containing a given polygon. Computing the convex hull of a set of points is known to be in $\Omega(n \log n)$,[2] which drops to $\Omega(n)$ if the set of points is sorted by coordinate or angle. There are well-known algorithms for computing the convex hull in optimal time, such as Graham scan or Andrew's monotone chain. For my implementation I have chosen the "contour polygon" algorithm, which is simple, fast and not very well-known

[1] For a proof, see [7].

[2] In the worst case—there are output-sensitive algorithms which perform slightly better in specific cases.

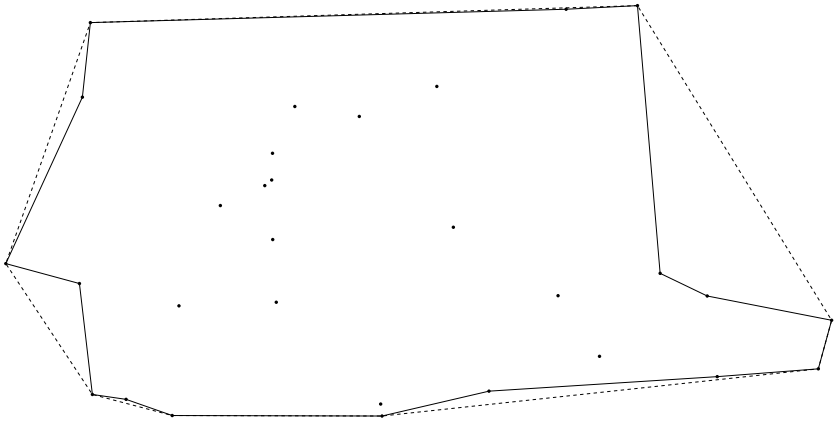


Figure 2.1: Contour polygon and convex hull

in the literature.[3] This algorithm proceeds in two steps. In the first step we create a "contour polygon" from a set of points sorted by their x-coordinate. In the second step, the contour is traversed to eliminated concave parts (see figure 2.1).

The contour polygon is created from the set of points by a double plane sweep. After sorting the points by x-coordinate we proceed with a plane sweep from left to right in order to create the left contour. For illustration, you can imagine the points of the set as nails sticking out of a board and the left contour as the result of blowing a long piece of thread against the nails from the left. In algorithmic terms, we use a very simple sweep status structure which stores only the minimum and maximum y-coordinate of the points added up to the current position of the sweep line. The sweep starts with the leftmost point in the set and then adds a point at the front of a double-ended queue if its y-coordinate is greater than the maximum y-coordinate so far or at the end of the queue if its y-coordinate is less than the minimum y-coordinate so far (see algorithm 1).

For the complete contour polygon, we repeat the sweep from right to left. The complete list of all points in the two parts of the contour may contain duplicates at the top and bottom, which we can eliminate in $O(1)$. There are also degenerate cases where points have the same x-coordinate or the same y-coordinate. In those cases the contour polygon may contain collinear edges, which is harmless. The contour polygon contains all points of the original set. Also, all vertices of the convex hull polygon are already vertices of the

[3]The original source for this algorithm is an unpublished technical report [6]. The only published reference I could find is in [9]. I have tested my implementation of the contour polygon against an implementation of Andrew's monotone chain algorithm borrowed from [4] and found that it slightly outperformed the monotone chain in most cases.

Algorithm 1 Left Contour

Input: A list P of points in the plane.
Output: A double-ended queue LC of the points of the left contour of P in counter-clockwise order.

```
1: Sort P in ascending order by x-coordinate.
2: Add the first point of P to LC.
3: Initialize minY and maxY with the y-coordinate of that point.
4: for all subsequent points p in P do
5:     if y-coordinate of p > maxY then
6:         maxY ← y-coordinate of p
7:         Add p at the front of LC.
8:     end if
9:     if y-coordinate of p < minY then
10:        minY ← y-coordinate of p
11:        Add p at the end of LC.
12:    end if
13: end for
```

contour polygon. In order to get the convex hull from the contour polygon we only have to eliminate all concave parts of the contour.

This process starts at the topmost vertex of the contour. Because it is an extreme point, the corner must be convex (we don't have to check it). We check the next corner for convexity by drawing a line from the first to the second vertex of the contour and determining whether the third vertex lies to the left of that line. If yes, the corner at the second vertex is convex and we proceed to the next vertex. If not, the corner is concave and we delete the second vertex, thereby replacing two edges with an edge from the first to the third vertex. In the general case, we may have to retrace our steps further and delete vertices until we find an edge which has our target vertex to the left of it (see algorithm 2).

While the algorithm does retrace some steps, the respective vertices are eliminated and never visited again, so the run time is $O(n)$. In total, the run time is $O(n \log n)$ if the set of points is not sorted and $O(n)$ if it is.

2.2 Minimum Bounding Box

Given the convex hull, there is an algorithm first described by Toussaint that computes the minimal bounding box in linear time. This algorithm is an example of the "rotating calipers" paradigm that can be applied to a variety

Algorithm 2 Convex Hull

Input: A list V of the vertices of a simple polygon in counter-clockwise order, starting at the topmost point.

Output: A list V of the vertices of the convex hull polygon in counter-clockwise order.

```
1: i ← 0
2: while i < size of V − 1 do
3:     if not v_{i+2} left of line from v_i to v_{i+1} then
4:         j ← i
5:         while j > −1 or not v_{i+2} left of line from v_j to v_{j+1} do
6:             j ← j − 1
7:         end while
8:         Delete vertices from v_{j+2} to v_{i+1} (inclusive).
9:         i ← j
10:    end if
11:    i ← i + 1
12: end while
```

of problems in algorithmic geometry.[4] Given the theorem that one side of the minimum bounding box of a set of points must be collinear with one side of the convex hull polygon of the set, one could compute the minimum bounding box in $O(n^2)$ by rotating the convex hull polygon to make each side collinear with the x-axis and constructing the axis-aligned bounding box from the extreme points. Shamos original idea was that instead of rotating the whole polygon, which takes $O(n)$ time, it is sufficient to find the antipodal points for each rotation, which is possible in constant time.

We first define a pair of lines of support as a pair of parallel directed lines of opposite direction such that the polygon lies wholly to the left of each line. A pair of antipodal points can then be defined as a pair of points of a polygon that admit a pair of lines of support. Finally, a set of calipers for the minimum bounding box problem is a set of two orthogonal pairs of lines of support. At the outset we compute the axis-aligned minimum bounding box of the polygon in $O(n)$ and initialize the calipers with lines collinear to the sides of that bounding box. We then calculate the minimum of the four angles between the lines of the calipers and the corresponding edges of the polygon. If there is more than one vertex that admits a given line, we pick

[4]The first application of the rotating calipers was for calculating the diameter of a convex polygon [12]. The expansion into an algorithmic paradigm (including the minimum bounding box) is due to [16]. Different applications of the paradigm were usefully collected in [11].

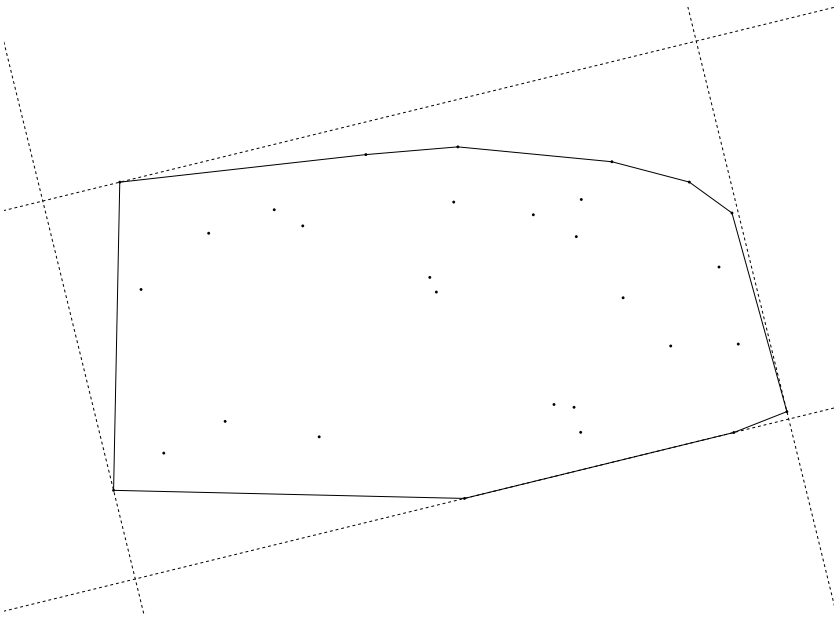


Figure 2.2: Convex hull with a set of calipers

the vertex that comes first in the counter-clockwise order of vertices. If that minimum is zero, we have a first candidate for the minimum bounding box.

For the algorithm it is convenient to define a custom datatype which holds the current total rotation of the calipers, the current set of antipodal points and the angles between line of support and polygon edge at that point. In the main loop of the algorithm we first update each antipodal point whose angle is zero by going to the next vertex of the polygon in counter-clockwise direction until the angle is greater than zero. We then determine the minimum of the four angles at the new antipodal points and rotate the calipers counter-clockwise by that angle. The line of support that yielded the minimum angle is now collinear with the edge of the polygon and we can generate a new bounding box candidate, which is added to the list (see figure 2.2). The process continues until the calipers have performed at least a 90 degree rotation, at which point no new candidates will be generated (see algorithm 3). The minimum-area bounding box can then be filtered from the list in $O(n)$.

For a convex polygon with n edges there are at most n iterations of the main loop. All calculations in one iteration can be performed in constant time, so the total time complexity is $O(n)$.

Algorithm 3 Rotating Calipers

Input: A convex polygon with vertices in counter-clockwise order.
Output: An oriented rectangle giving the minimum-area bounding box of that polygon.

```
Initialize calipers by calculating the axis-aligned bounding box of the input polygon.
if any angle = 0 then
    Add box to set of bounding boxes.
end if
while total rotation < π/2 do
    Update antipodal points until all angles > 0.
    Rotate calipers by minimum angle at antipodal points.
    Generate bounding box based on total rotation and antipodal points.
    Add box to set of bounding boxes.
end while
return the minimum-area bounding box in the set
```

2.3 Rectangles in a Circle

By replacing a polygon with its minimum bounding box we reduce the nesting problem to the problem of packing rectangles in a circle. Packing problems involving simple geometric shapes are a source of great fascination – intuitively one expects optimal solutions to be simple and regular, but in fact they are often quite complex and irregular. The closest analogue to our rectangles-in-circle problem is the squares-in-circle problem (see figure 2.3 and [14] for more cases).[5]

When looking at the list of optimal solutions for the squares-in-circle problem for cases $n = 1$ to $n = 35$, roughly a third of the solutions have non-axis-aligned squares. Of the solutions in which all squares are axis-aligned, roughly two-thirds conform to a pattern of rows or columns, which therefore seems a promising heuristic. Clearly, some sort of heuristic is required for a pragmatic approach since optimal solutions are too hard to find. Proving them to be optimal is even more difficult; according to [14] only for the cases $n = 1$ and $n = 2$ has the optimality of the packing been rigorously proven. One interesting feature of the rows-or-columns pattern is that it is not always symmetric, i.e. there may be rows of different lengths at the top and bottom or columns of different length on the left and right. This feature can be replicated in an algorithm by varying the starting position.

[5] One reason the rectangles-in-circle-problem hasn't seen a lot of research is probably that there is no clear conception of a "unit rectangle".

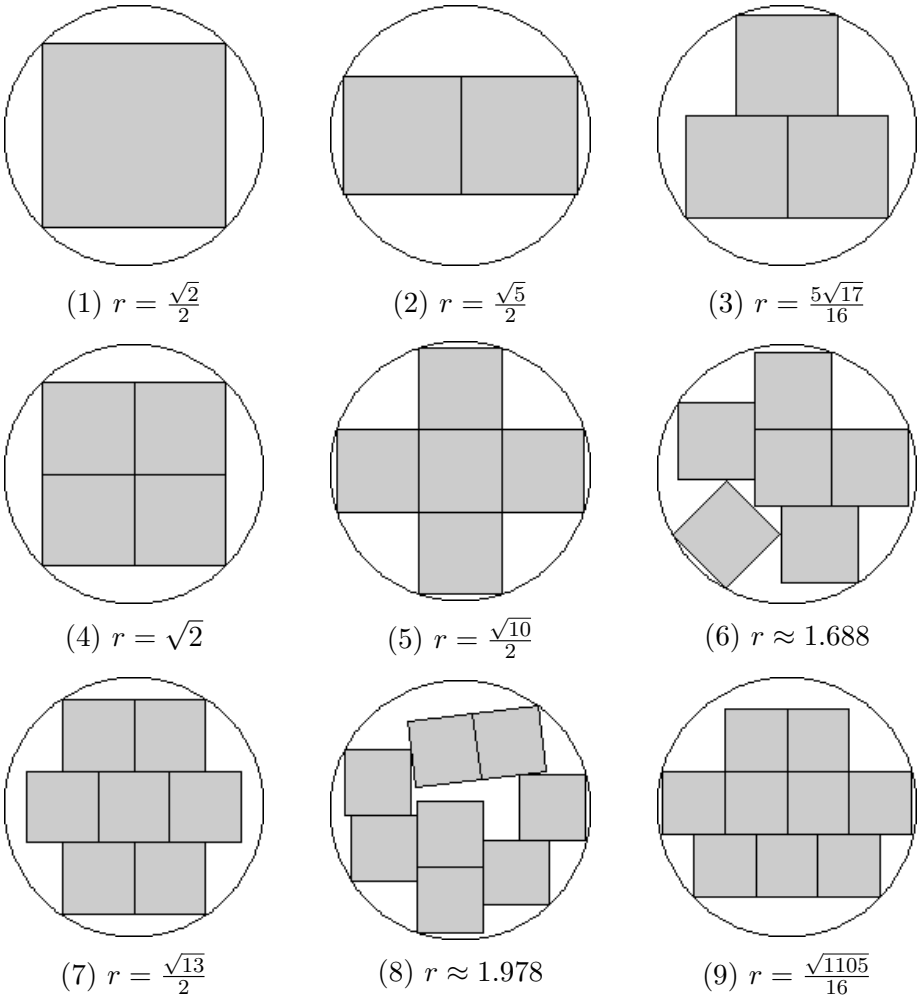


Figure 2.3: Squares in circles

I have tried to corroborate the quality of the rows-or-columns heuristic by testing it against the optimal solutions presented in [14]. For the starting position I have used a rasterization of 1/1000 of the unit length. In packing squares it makes no difference whether we do it in rows or columns. Choosing rows, we start the first row at the bottom of the circle and then keep adding rows until we reach the top of the circle. The number of squares that can be placed at a certain position is equal to the floor of the minimum chord length at the bottom and top of the row. Running this algorithm with the radii given in [14], we get an average n_{max} of 16.914 as oppposed to 18 for the optimal solutions. So, as expected, we lose one square relative to the optimal solution, but rarely more than one. If the heuristic is given a little more space by increasing the radii of the enclosing circles by just one percent,

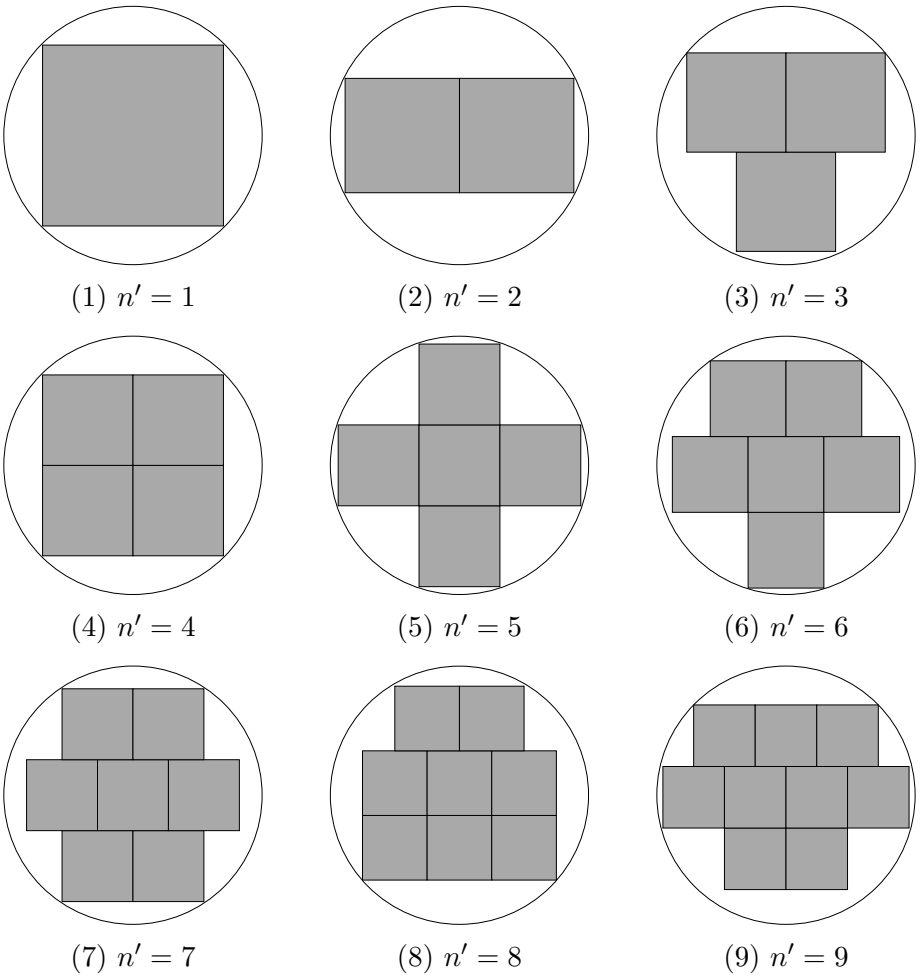


Figure 2.4: Packing unit squares ($r' = 1.01r$)

the heuristic is on par with the optimal results ($n_{max} = 17.943$).

Putting the rows-or-columns pattern into practice for packing preforms requires one more modification. Because we have to guarantee a minimum distance between preforms, the minimum bounding boxes have to have the minimum distance with respect to each other. This could be done by calculating an offset curve for the box, which has the shape of a rectangle with rounded corners. See figure 2.5, where the offset curve is represented by the dashed lines. But since the boxes are arranged in rows or columns, we can guarantee the minimum distance with respect to each other by simply spacing them apart on the x- and y-axis. The only non-trivial part is guaranteeing the minimum distance between box corners and table edge, which can be done by a slight modification to the calculation of the chord length.

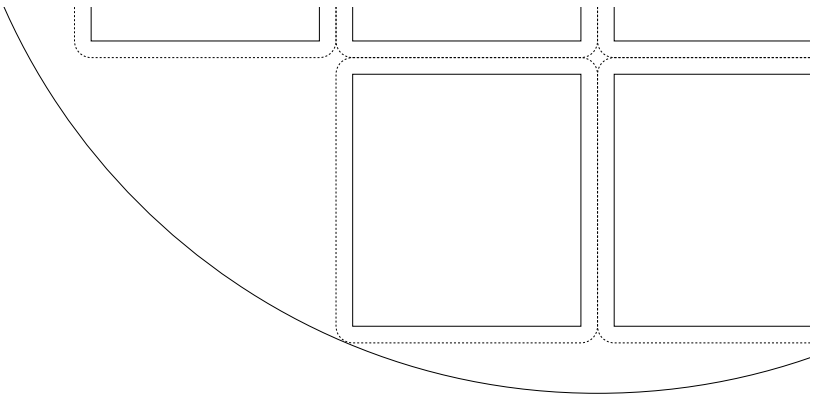


Figure 2.5: Box packing

Illustrating again for the case of rows, if we take the position p of a row as the distance from the circle center of the lower edge of the offset curve of the bounding box, h as the height of the bounding box, r as the radius of the circle, d as the minimum distance between preforms, and $cl(x, y)$ as the function giving the length of a chord of a circle of radius x whose Euclidean distance from the center of the circle is y, the available length $l(p)$ can be gotten from the following equation:

$$l(p) = \min(cl(r - d, p + d) + 2d, cl(r - d, p + h + d) + 2d)$$

Dividing this value by the width of the offset curve, which is the width of the bounding box plus $2d$, and rounding down gives the number of boxes that can be placed in that row while keeping the minimum distance to each other and the table edge.

Chapter 3

Circle Packing

In circle packing, the preform is approximated by its smallest enclosing circle. As in box packing, there are two subproblems:

1. calculating the smallest enclosing circle efficiently and
2. maximizing the number of placed circles.

3.1 Smallest Enclosing Circle

The smallest enclosing circle of a set of points is unique. It is either defined by two points diametrically opposite of each other or by three points.[1] A brute-force algorithm calculating the smallest enclosing circle has time complexity $O(n^4)$. An $O(n \log n)$ algorithm was presented by Skyum in [13].[2] Asymptotically, the best approach is an expected-time $O(n)$ algorithm using randomized incremental construction presented by Welzl in [17].

For randomized incremental construction, let $p_1, \ldots, p_n$ be a random permutation of a set of points. Let $P_i := \{p_1, \ldots, p_i\}$. We then have the following lemma concerning the smallest enclosing circle C_i of P_i:[3]

Lemma 1 *Let $2 < i \leq n$, and let P_i and C_i be defined as above. Then we have*

[1] There may be more than three points on the circle, but any triple is sufficient to define it.

[2] Skyum references some $O(n)$ linear programming algorithms applicable to the smallest enclosing circle problem and claims that "the involved constants hidden in $O(n)$ are large". I have experimentally determined that the constant in Welzl's algorithm (counting loop iterations) varies roughly between 2 and 20, which isn't too bad. Nevertheless, Skyum's $O(n \log n)$ may be faster for small n.

[3] For proofs, see [17] and [2]. The following presentation is mostly adapted from [2].

(i) If $p_i \in C_{i-1}$, then $C_i = C_{i-1}$,

(ii) If $p_i \notin C_{i-1}$, then p_i lies on the boundary of C_i.

We start out by calculating the smallest enclosing circle of the first two points p_1 and p_2. We then add points one-by-one, maintaining the smallest enclosing circle C_i. Adding a point will either leave the circle unchanged or result in a new circle that has the added point on the boundary. In the first case, we go to the next point, in the second case we try to calculate the smallest enclosing circle of P_i with p_i is on the boundary.

Algorithm 4 Smallest Enclosing Circle

Input: A list P of n points in the plane.
Output: The smallest enclosing circle C of P.

```
1: Calculate a random permutation p_1, ..., p_n of P.
2: Generate the smallest enclosing circle C_2 of p_1 and p_2.
3: for i ← 3 to n do
4:     if p_i ∈ C_{i-1} then
5:         C_i ← C_{i-1}
6:     else
7:         C_i ← smallest enclosing circle of P_i with p_i on the boundary
8:     end if
9: end for
10: return C_n
```

For calculating the smallest enclosing circle of a set of point P with a point q on the boundary, we re-apply the above approach. We randomize the set P and start with the smallest enclosing circle of p_1 and q.[4] We add points from P until we meet a point p_i which isn't in the circle. We now know that p_i and q are on the smallest enclosing circle of P_i.

In order to calculate the smallest enclosing circle of a set of points with two points on the boundary, we start over with the smallest enclosing circle of the two points on the boundary, adding points. Any point that's not in the the circle gives us a third point we need to construct the next smallest enclosing circle until all points in the set are incorporated.

Algorithm 6 runs in $O(n)$ because every iteration of the loop takes constant time. The running time of algorithm 5 is $O(n)$ as long as we don't consider calls to algorithm 6. In the worst case there can be $O(n)$ such calls.

[4]We can also use a previous randomization of the input if the algorithm is called as a subroutine.

Algorithm 5 Smallest Enclosing Circle With Point on the Boundary

Input: A list P of n points in the plane and a point q on the boundary of the smallest enclosing circle of P.

Output: The smallest enclosing circle C of P with q on the boundary.

```
1: Calculate a random permutation p_1, ..., p_n of P.
2: Generate the smallest enclosing circle C_1 of p_1 and q.
3: for i ← 2 to n do
4:     if p_i ∈ C_{i-1} then
5:         C_i ← C_{i-1}
6:     else
7:         C_i ← smallest enclosing circle of P_i with p_i and q on the boundary
8:     end if
9: end for
10: return C_n
```

Algorithm 6 Smallest Enclosing Circle With Two Points on the Boundary

Input: A list P of n points in the plane and two points q_1 and q_2 on the boundary of the smallest enclosing circle of P.

Output: The smallest enclosing circle C of P with q_1 and q_2 on the boundary.

```
1: Calculate a random permutation p_1, ..., p_n of P.
2: Generate the smallest enclosing circle C_0 of q_1 and q_2.
3: for i ← 1 to n do
4:     if p_i ∈ C_{i-1} then
5:         C_i ← C_{i-1}
6:     else
7:         C_i ← circle with p_i, q_1 and q_2 on the boundary
8:     end if
9: end for
10: return C_n
```

But what is the expected number of calls? Using backwards analysis, we consider the situation after having constructed C_i. What is the probability that this construction was expensive, i. e. resulted from a call to algorithm 6? It is the same as the probability that the smallest enclosing circle C_i changes (decreases in size) if we remove a random point from P_i. This probability is at most $\frac{2}{i}$ if the C_i is determined by three points on the boundary and $\frac{1}{i}$ if if it is determined by two points on the boundary. So the expected running

time for adding point p_i is at most

$$\frac{i-2}{i} \cdot O(1) + \frac{2}{i} \cdot O(i) = O(1),$$

giving a total runtime of $O(n)$ for algorithm 5. The same form of argument can be applied to algorithm 4, so its expected running time is $O(n)$ as well.

3.2 Circles in a Circle

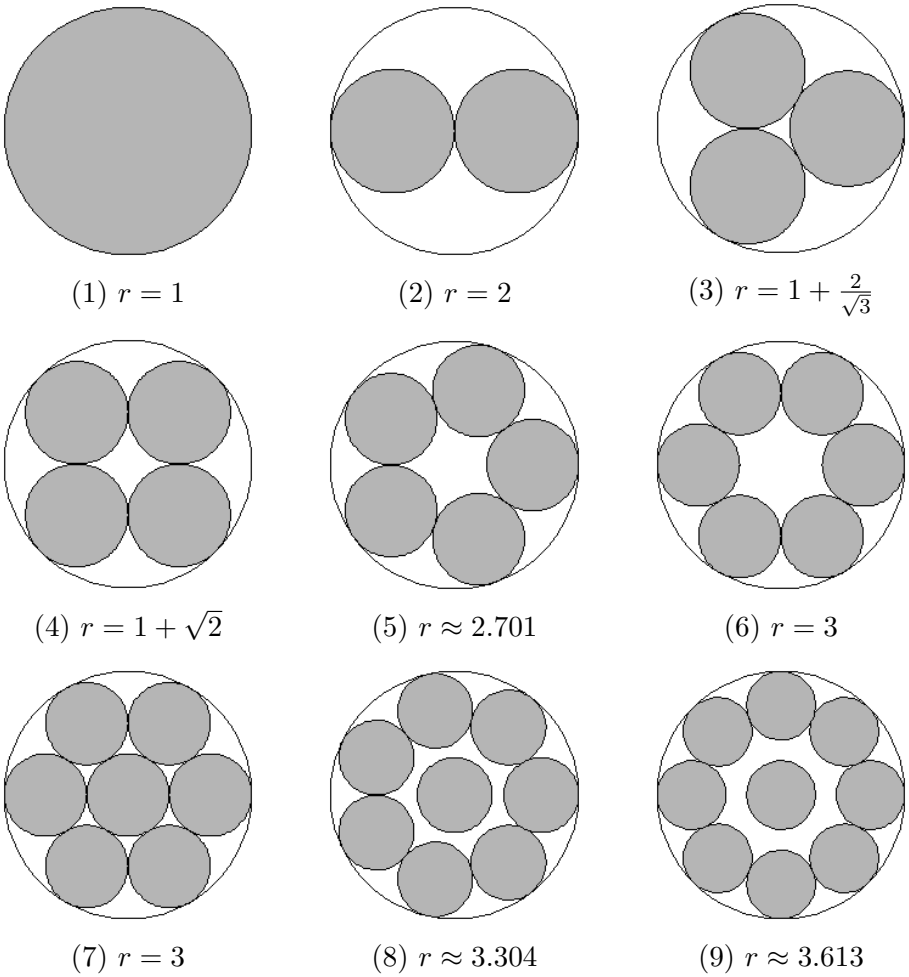
Packing a number of unit circles into a smallest enclosing circle is another popular packing problem. Once again, optimal solutions exhibit a high degree of irregularity.[5] It's clear that in the infinite plane the hexagonal pattern is the packing with the highest density,[6] but unlike the rows-and-columns pattern in box packing the hexagonal pattern is optimal in very few cases. This is partly due to the fact that the pattern is rigid, i.e. there is no degree of freedom allowing it to adapt to the shape of the enclosing circle. In principle, we could solve the circle packing problem by simply calculating the ratio between the radius of the smallest enclosing circle and the table radius and taking the optimal solution from the database in [15]. If we want to proceed computationally, as in box packing, we have to choose a heuristic because finding the optimal solution is too expensive, especially relative to the fact that for most preforms a circle is a very poor approximation.

A priori, the hexagonal packing seems like a good heuristic, because it has the highest density and also conforms well to the circular boundary. As in box packing, I have tried to estimate the quality of the heuristic experimentally. There are a few cases (e.g. $n = 4$) where a regular pattern is better than the hexagonal pattern and so I have included that pattern in the heuristic. The hexagonal packing of the plane has a period of 2 unit lengths in the horizontal dimension and a period of $2\sqrt{3}$ unit lengths in the vertical dimension. The regular packing has a period of 2 unit lengths in both dimensions. If we use the same parameters as for the squares-in-circle problem, i.e. a slightly larger radius of $r' = 1.01r$ and a rasterization of $\frac{1}{1000}$ of the unit length in both dimensions, the average n_{max} for cases $n = 1$ to $n = 35$ is 16.714 versus 18 for optimal solutions. Since most of the patterns generated by the algorithm were symmetric, I have also experimented with a one-dimensional search along the horizontal and vertical axes of symmetry, yielding an average n_{max} of 16.6857.[7]

[5] See figure 3.1. These cases are taken from [3]. See [15] for cases up to $n = 2600$.

[6] The density of a hexagonal packing of circles in the plane is $\frac{\pi\sqrt{3}}{6} \approx 0.9069$.

[7] Interestingly, there was only one case in the interval up to $n = 35$, namely $n = 27$



(1) $r = 1$ (2) $r = 2$ (3) $r = 1 + \frac{2}{\sqrt{3}}$

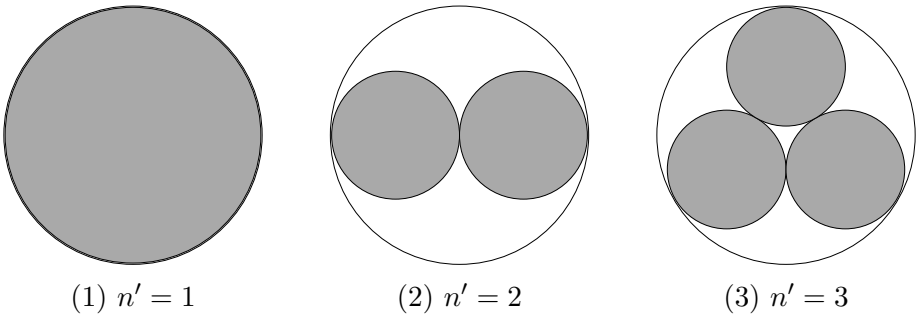
(4) $r = 1 + \sqrt{2}$ (5) $r \approx 2.701$ (6) $r = 3$

(7) $r = 3$ (8) $r \approx 3.304$ (9) $r \approx 3.613$

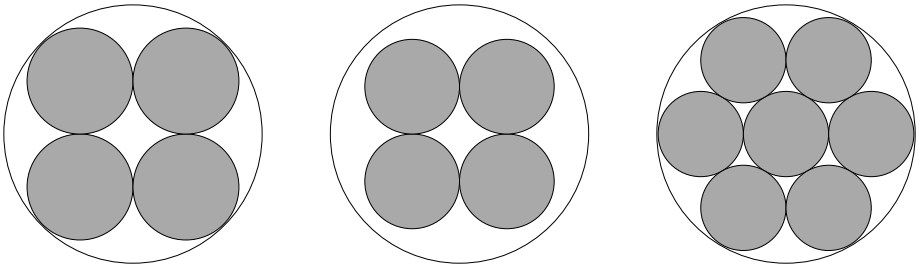
Figure 3.1: Circles in circle

For packing preforms, we calculate the smallest enclosing circle of the preform outline polygon. The part clearance can be taken into account by simply adding 22 mm to the radius of that circle. We then generate sufficiently large hexagonal and regular packing of this circle and move a circle representing the table edge over this packing, filtering all positions entirely contained in it.

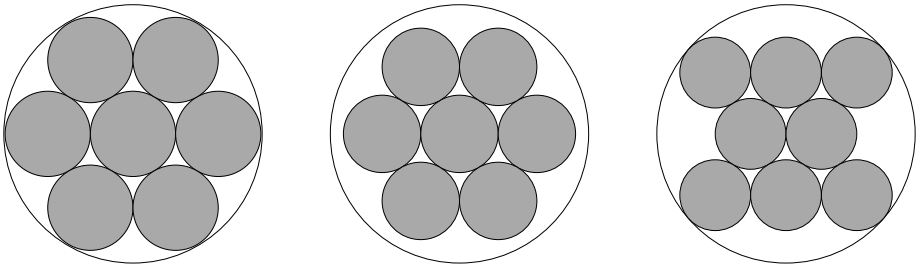
where there is an asymmetric hexagonal pattern that is better than the best symmetric pattern.



(1) $n' = 1$ (2) $n' = 2$ (3) $n' = 3$



(4) $n' = 4$ (5) $n' = 4$ (6) $n' = 7$



(7) $n' = 7$ (8) $n' = 7$ (9) $n' = 8$

Figure 3.2: Packing unit circles ($r' = 1.01r$)

Chapter 4

Irregular Shape Nesting

While in box and circle packing the positioning of polygons can at least partly be based on closed-form geometric calculations, finding the best possible arrangement of arbitrary irregular shapes necessarily involves some degree of experimentation. I will use the term "nesting strategy" for the ensemble of choices that have to be made in designing any particular algorithm. I will start by laying out a simple "one-by-one" nesting strategy. It places a polygon in the center of the circular area and then adds polygons one-by-one, placing them as closely as possible to the polygons already placed until no more legal positions are available.

Deciding at what position to place the next polygon in line involves at least three different subproblems:

1. Defining a search space, i.e. a set of positions to evaluate
2. Checking the legality of a position (non-intersecting, minimum distance)
3. Evaluating a position (quality of nesting)

While subproblems one and three present themselves differently for different nesting strategies, the second is basic to all strategies, so I will discuss it first.

4.1 Offset Curve

Compared to the way nesting problems are usually formulated, checking the legality of a position is a little different for our problem because not only should polygons not intersect, they also have to have a minimum distance

to each other. Checking this distance requirement can be done in one of two ways: One could calculate the distance between two polygons directly and check it against the minimum distance or one could calculate an offset curve and then check offset curves for intersection. I have chosen the second approach because it reduces the problem of checking the minimum distance to the familiar problem of checking polygons for intersection.

The concept of an offset curve is based on the mathematical concept of a parallel curve, which generalizes the concept of the parallel of a line in the Euclidean plane. There are two ways to define the parallel curve to a given curve:

1. the envelope of a family of congruent circles centered on the curve, or
2. the curve whose points are at a fixed normal distance to the curve.

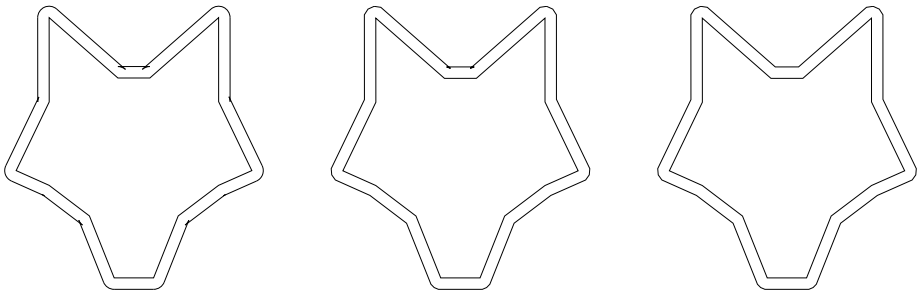
These definitions are equivalent for smooth curves; for non-smooth curves, the second definition has to be slightly amended.[1] The envelope of a family of curves is a curve that is tangent to each member of the family at some point. Given a certain normal distance d, e.g. our part clearance of 22 mm, the parallel curve to a polygon with that distance d is the envelope of a family of congruent circles with radius d centered on the edges of the polygon. It is composed of two kinds of elements:

1. for each edge of the polygon, there is a right-hand parallel edge with normal distance d;
2. for each vertex of the polygon, there is a circular arc with radius d and angle corresponding to the angle between the normals of the edges connected by the vertex.

Since we only have to traverse the vertices of the polygon once to calculate these elements, an algorithm to calculate the parallel curve of a polygon runs in linear time. The difference between a parallel curve and an offset curve is that the offset curve only contains points with *minimum* distance d to each point on the circumference of the polygon. For convex polygons, the parallel curve is the offset curve. For non-convex polygons, self-intersecting parts of the parallel curve have to be clipped in order to get the offset curve (see figure 4.1 for the offset curve of the coyote head preform).

Although it would be possible to use exact offset curves for nesting, the algorithmic treatment is simpler when circular arcs are approximated by

[1] See [18], section "Parallel curve to a curve with a corner" for the concept of a "normal fan".



(1) Parallel curve (2) Polygonized parallel curve (3) Offset curve

Figure 4.1: Offset curve

polygonal chains. In order to maintain the minimum distance, these chains have to consist of line segments tangent to the arcs. The grain of the polygonization can be controlled by choosing a value for the maximum normal distance between a point on the arc and the tangent edge. In practice, a value of 1 mm gives a good balance between fit and complexity. The number of vertices required to polygonize an arc is determined by the maximum angle that can be covered by introducing a single vertex. Letting r be the radius of the arc and δ the maximum normal distance, the maximum angle α that can be covered by a vertex is:

$$\alpha = 2 \arcsin \frac{\sqrt{(r+\delta)^2 - r^2}}{r+\delta}$$

It's then straightforward to calculate the vertices that have to be introduced. On a convex corner the parallel edges are extended to the first and last new vertex; on a concave corner they are shortened.

Since the algorithm runs only once and the number of edges is usually fairly small (< 100), we have used a simple brute force algorithm to eliminate self-intersections. We determine a vertex on the convex hull of the polygonized parallel curve, which is guaranteed to be an outside vertex. Starting from this vertex, we check each edge in turn against the following edges (excluding consecutive edges) until an intersection is found. Since we have started from an outside vertex, the part of the curve between the first intersecting edge and the second intersecting edge cannot be part of the outline. We can therefore replace the end point of the first intersecting edge and the starting point of the second intersecting edge with the point of intersection and discard the intervening edges. We repeat this process until no more intersections are found. In theory, there can be up to $O(n)$ self-intersections. If array-based lists are used, eliminating an intersection itself is $O(n)$, so

the total run time should be in the area of $O(n^3)$. In practice, the absolute run time of the algorithm for our set of test cases is negligible (a few milliseconds).[2]

4.2 No-Fit Space

Checking the legality of the position of a polygon involves

1. checking whether the polygon fits inside the circular nesting area
2. checking whether the polygon has the minimum distance to the polygons already placed.

The first part is easy: We simply check the distance of each vertex of the offset curve polygon to the table center against the radius of the table. The second part involves checking whether any two offset curves intersect.

Checking offset curves, which are simple polygons, for intersection can be reduced to the existence problem of line segment intersection, which is solvable in $O(n \log n)$ with a simple sweep line algorithm. Another elegant tool for checking polygons for intersection is the no-fit polygon. One advantage in our case is that there are only two possible orientations of the offset curve polygon. It either has normal orientation or it is rotated by 180°. So, in order to check whether two arbitrarily placed offset curve polygons intersect we would only have to calculate the no-fit polygon for two instances of the polygon with the same orientation and the no-fit polygon for two instances of the polygon with different orientation once and then check whether the vector expressing the relative position of the two polygons is inside or outside of the no-fit polygon.

Unfortunately, calculating the no-fit polygon in the general case of two possibly non-convex polygons is a very complex and difficult problem, which is beyond the scope of this thesis.[3] I have therefore chosen a different approach which is much simpler and harnesses many of the same benefits. As pointed out, in our case all intersection checks can be reduced to the two basic cases of two polygons with the same or different orientation placed in a certain relative position. Therefore, once we have chosen a certain rasterization of the search space, we only have to calculate the total set of all

[2] In principle, it would be possible to use a variant of the Bentley-Ottman algorithm to report self-intersections in $O(n \log n)$. But reporting them isn't enough, we also have to eliminate them, which would be complicated by the fact that the edges have been sorted by x-coordinate and are no longer in the order in which they appear in the curve. So, for practical reasons, we haven't pursued this approach for now.

[3] See [10] for some recent research.

relative positions at which the polygons intersect once when the nesting algorithm is initialized. We generate a search space with the given rasterization that's large enough to contain all possible positions at which the polygons could intersect and then do intersection checks for these relative positions. The resulting set of relative positions at which the polygons intersect I have called the "no-fit space" for the two polygons with the given orientation (see algorithm 7)

Algorithm 7 No-Fit Space

Input: Two polygons, a rasterization.

Output: A set of vectors expressing relative positions at which the two polygons intersect.

1: Choose a reference point for each polygon, e.g. the bottom-left corner of its bounding box.

2: Generate a search space with the given rasterization which is large enough to contain all relative positions at which the polygons could intersect (for example, a rectangular space with twice the width and height of the bounding box). Elements in the search space are vectors by which one polygon is translated relative to the other. The vector $(0, 0)$ expresses the situation where the reference points of the polygons are superimposed (i.e. have the same coordinates).

3: Do intersection checks for each vector in the search space. The vectors at which the polygons intersect constitute the no-fit space for these two polygons.

There are various possible approaches for testing simple polygons for intersection, for example decomposing them into convex polygons and then testing these convex sub-polygons with an $O(n)$ sweep-line algorithm, but the simplest approach is reducing the problem to the problem of line segment intersection. For this problem, we could adapt the well-known Bentley-Ottman sweep-line algorithm. However, the simplest form of this algorithm requires that line segments not have the same x-coordinate and that there are no vertical line segments. Ensuring that these requirements are met or removing them is somewhat arduous in our case because every line segment shares an endpoint with at least one other line segment and there often are vertical line segments.[4] Because there are usually far fewer than 100 line segments in the polygons we use, we have tried a brute-force line segment intersection

[4]As a first shot, meeting the requirement of no vertical lines could be solved by computing the minimum deviation from the vertical of any edge in the polygon and then rotating the polygon by half that angle. Removing the requirement of different x-coordinates (and y-coordinates, by the way) could be effected by making sure that end point events of

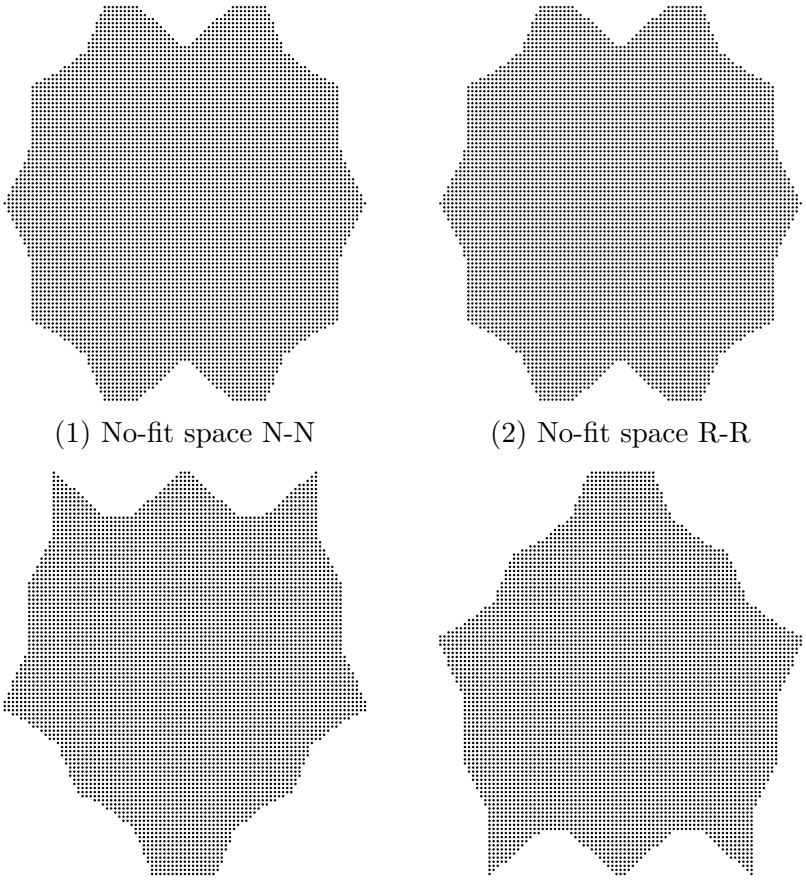
algorithm which tests each line segment in one polygon against each line segment in the other polygon. From the standpoint of absolute (not asymptotic) runtime, the difference in run-time between this $O(n^2)$ algorithm and the $O(n \log n)$ sweep-line algorithm is mitigated by the fact that for the brute-force algorithm, n is the number of line segments in one polygon, while for the sweep line algorithm n is the number of line segments in both polygons. Also, the intersection checks are run only once when the nesting algorithm is initialized. In practice, the total run time for calculating the no-fit spaces by brute-force for a typical preform polygon and a reasonably fine-grained rasterization is a few tenth of a second, which is entirely sufficient for our purposes.

There are a total of four no-fit spaces for each pair of polygons, depending on whether one or the other has normal orientation or is rotated by 180°. I have used acronyms to label these spaces and all spaces that can be constructed from them: *N-N* for both polygons in normal orientation, *N-R* for the fixed polygon in normal orientation and the "orbiting" polygon rotated, and so on. Only two of those spaces have to be calculated explicitly (*N-N* and *N-R*), the other two can be constructed by point reflection (*R-R* from *N-N*, *R-N* from *N-R*). See figure 4.2 for a graphical representation of the no-fit spaces for the coyote head preform.

4.3 Fit Space

The no-fit spaces give us a method to calculate, based on a certain configuration of preforms already on the table, the positions at which additional polygons *cannot* be placed. The next task is to construct a subset of the set of positions at which polygon can be placed in order to evaluate them for the placement of an additional polygon. Since for performance reasons we are not using any form of branching or backtracking, we have to evaluate a position based only on properties of the configuration already on the table in conjunction with the prospective placement. In this situation it seems self-evident that an optimal placement can only be one in which the next polygon to be placed touches at least one of the polygons already on the table. In terms of the no-fit spaces, these would be the positions just "around" the no-fit spaces. In calculating the no-fit spaces, we have, in effect, already

line segments are processed before starting point events. It then shouldn't matter in what (temporal) order new line segments starting at the same x-coordinate are inserted, because no intersections will be missed. We don't have to process intersection events because we're only checking for the existence of intersections, not reporting them, so the algorithm terminates after finding an intersection.



(1) No-fit space N-N (2) No-fit space R-R

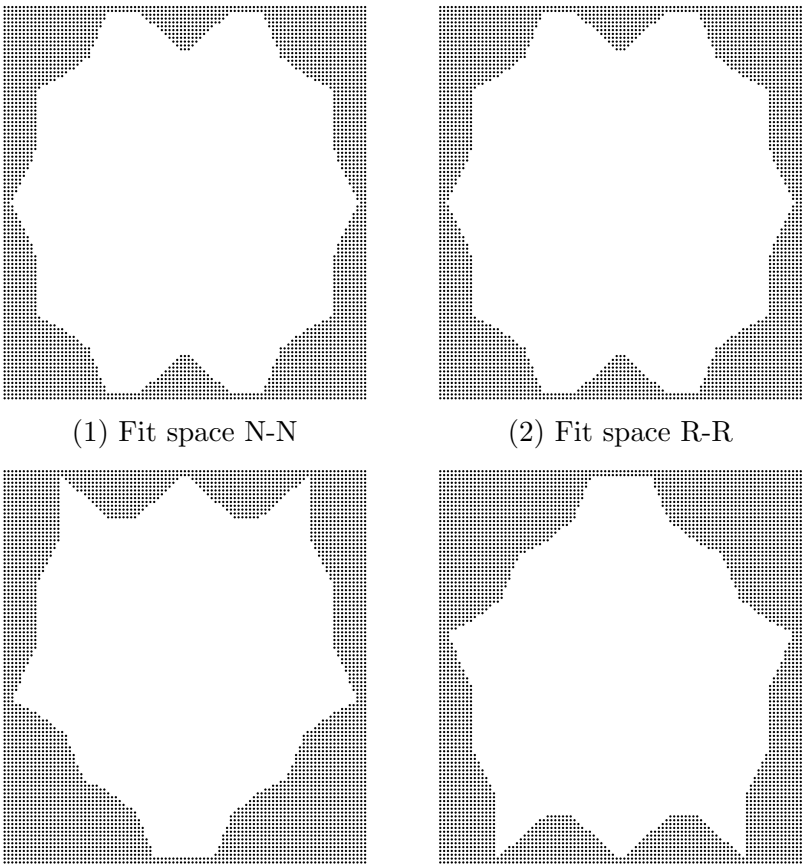
(3) No-fit space N-R (4) No-fit space R-N

Figure 4.2: No-fit spaces

calculated one version of these "fit spaces", which are simply the positions in the search space we used for calculating the no-fit spaces at which the polygons didn't intersect, or, in other words, the negative spaces of the no-fit spaces with respect to that search space. Figure 4.3 shows these negative space for a search space based on the bounding boxes of the polygons.

Obviously, these fit spaces are far from optimal; they contain too many points that we know won't lead to an optimal placement. The best way to limit these spaces without using the full no-fit polygon is to calculate the no-fit polygon for the convex hull polygons. In comparison to the no-fit polygon for arbitrary polygons, the no-fit polygon for two convex polygons is relatively easy to calculate using the sorted-edges approach.

For calculating the no-fit polygon of two convex polygons, we hold one polygon in a fixed position with the other polygon orbiting it in such a way that the orbiting polygon touches the fixed polygon. We choose reference



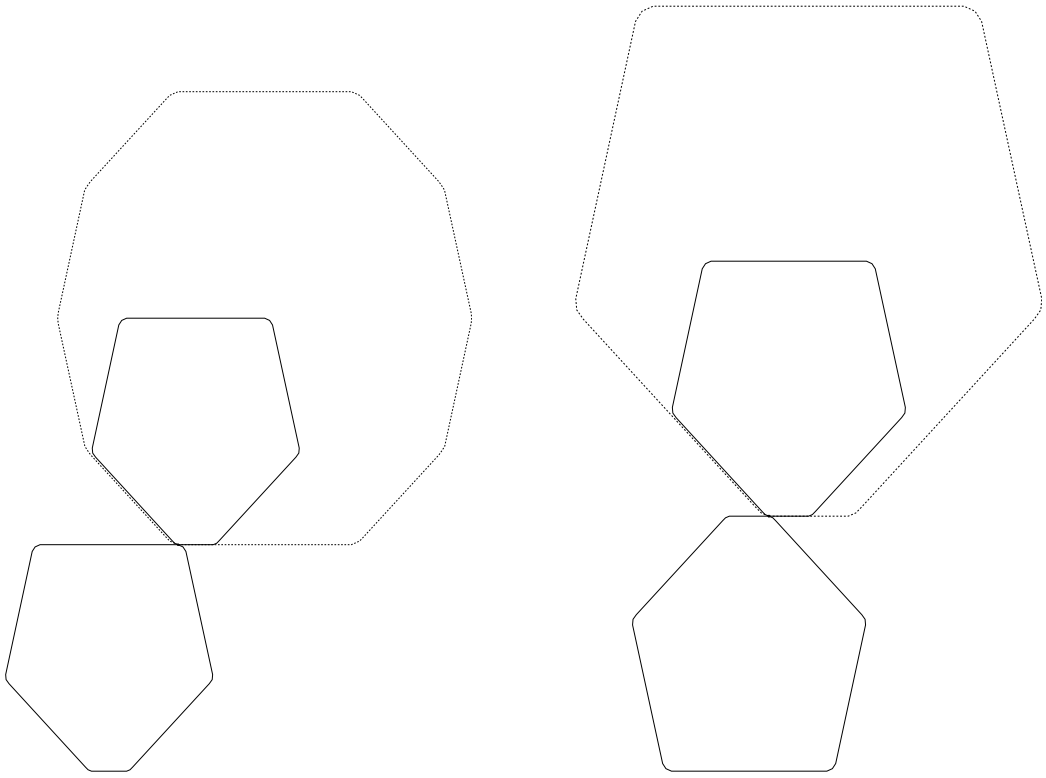
(1) Fit space N-N (2) Fit space R-R

(3) Fit space N-R (4) Fit space R-N

Figure 4.3: Fit spaces

points for each polygon. The fixed polygon's reference point is its lowest vertex. If there is more than one lowest vertex, we choose the left-most of these vertices. The orbiting polygon's reference point is its uppermost vertex. If there is more than one, we choose the right-most of them. The no-fit polygon then results from the path of the orbiting polygon's reference point as it slides along the circumference of the fixed polygon (see figure 4.4). The easiest way to calculate it is by the sorted-edges approach.[5] We make a list containing all the edges of the fixed polygon as directed line segments oriented in counter-clockwise direction, and all the edges of the orbiting polygon as directed line segments oriented in clockwise direction. We then sort this list in ascending order by the angle the edges make with the x-axis. Finally, we append the line segments to each other in this order, starting from the

[5]This algorithm was first described in [5]. A more formal treatment is contained in [8]. See also [10] for useful discussion.



(1) N-N (2) N-R

Figure 4.4: No-fit polygons of convex hull polygons

reference point of the fixed polygon. Since all these operation except sorting run in linear time, the algorithm has a run time complexity of $O(n \log n)$.

Strictly speaking, only the reference point of the fixed polygon is required to calculate the no-fit polygon. But we need the reference point of the orbiting polygon in order to apply the no-fit polygon as a test of whether the two polygons placed in a certain relative position intersect. Generally speaking, the no-fit polygon that is the result of the algorithm is located in the coordinate system of the fixed polygon. If we want to determine whether an instance of the orbiting polygon at a given relative position to an instance of the fixed polygon intersects it, we have to calculate the position of the orbiting polygon's reference point at this relative position. What makes this a little confusing is the fact that in order to express the relative position of two polygons, we also have to select arbitrary reference points for them, which are not identical to the reference points used in calculating the no-fit polygon. In our case, we have used the bottom-left corner of a polygon's axis-aligned bounding box as its general reference point. If we let r_{fixed} be

Algorithm 8 No-Fit Polygon

Input: Two polygons.

Output: The no-fit polygon and the reference points of the two polygons.

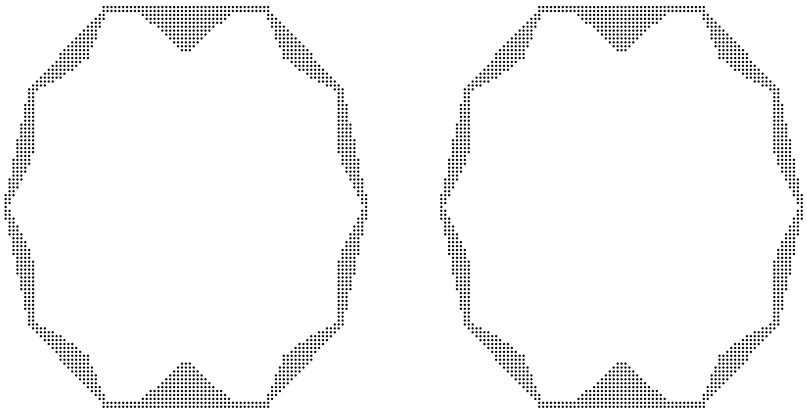
1: Select the fixed polygon's lowest vertex as its reference point.
2: Select the orbiting polygon's uppermost vertex as its reference point.
3: Make a list containing all the edges of the fixed polygon as directed line segments in counter-clockwise direction plus all the edges of the orbiting polygon oriented in clockwise direction.
4: Sort this list in ascending order by the angle the edges make with the x-axis.
5: Append the line segments to each other in this order, starting from the reference point of the fixed polygon.

the reference point of the fixed polygon as used in the no-fit polygon algorithm, $v_{position}$ be the vector expressing the relative position of the polygons with respect to their general reference points and $v_{reference}$ the vector from the orbiting polygon's general reference point to its reference point as used in the no-fit polygon algorithm, we get the position of the orbiting polygon's reference point $r_{orbiting}$ by the equation

$$r_{orbiting} = r_{fixed} + v_{position} + v_{reference}$$

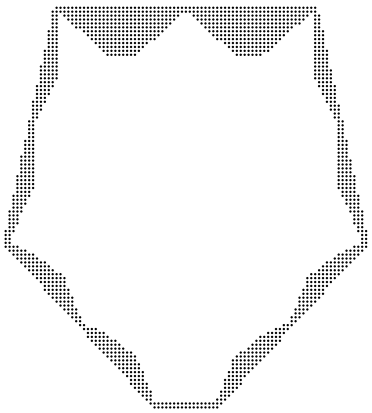
In order to determine whether the polygons intersect at this position, we only have to check whether the no-fit polygon contains $r_{orbiting}$. Figure 4.5 shows the improved fit spaces based on the convex no-fit polygon.

In the one-by-one nesting strategy I'm currently outlining, the situation defining the search space of each step of the algorithm is this: There already is an arrangement of one or more preforms on the table and we are looking for the best placement of the next one. It's clear that we need two search spaces: one for placing a polygon in normal orientation, one for placing a rotated one. Loosely speaking, we add up all the fit-spaces and subtract the no-fit spaces. In order to be more efficient, these operations should be performed incrementally as the arrangement of preforms grows. For this, we have to keep the union of the fit spaces and the union of the no-fit spaces from the last step of the algorithm and for each polygon added to the configuration, add all positions in the fit space that are not in the cumulative no-fit space, subtract all positions in the new no-fit space and update the cumulative no-fit space. See algorithm 9 for the operations necessary to incrementally construct a search space for placing a polygon in normal orientation. For placing a polygon in rotated orientation, the respective fit and no-fit spaces

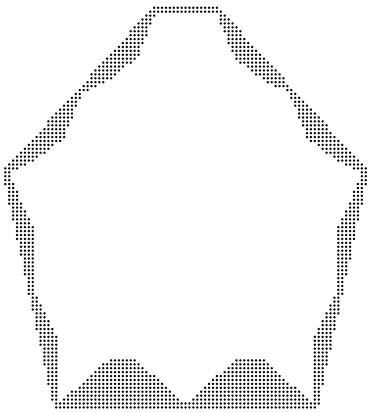


(1) Fit space N-N

(2) Fit space R-R



(3) Fit space N-R

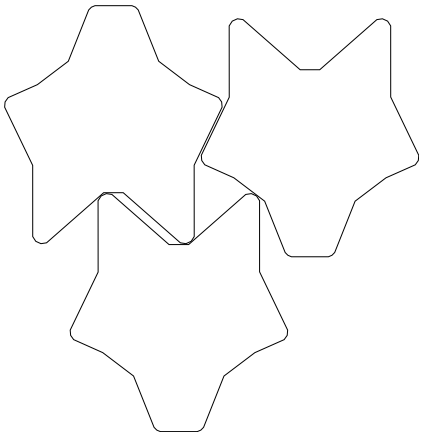


(4) Fit space R-N

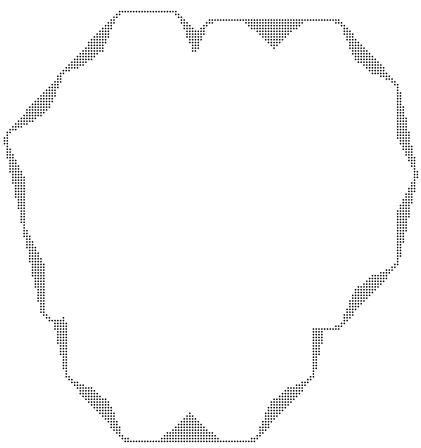
Figure 4.5: Improved fit spaces based on convex no-fit polygon

have to be substituted.

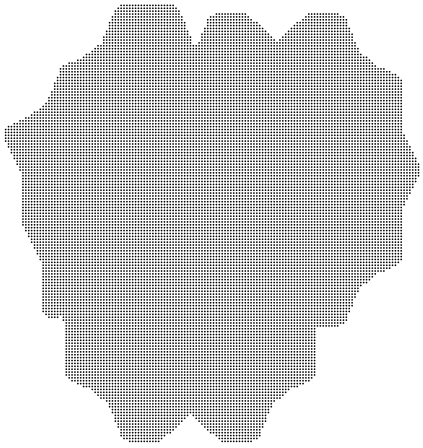
In order to perform these operations, we obviously need some data structure that supports set-theoretic operations. We have chosen to implement the elements of search spaces as integer vectors because checks for equality can be unreliable for floating point numbers. In order to perform insertion and removal of elements efficiently, we have used binary search trees, which are $O(\log n)$ for each operation on a set of n elements. If n is the number of elements in the search space, one incremental step of adding a polygon placement with a fit space of m elements has a run time of $O(m \log n)$. Figure 4.6 shows the search spaces and cumulative no-fit spaces for placing a polygon of arbitrary orientation next to an arrangement of three polygons.



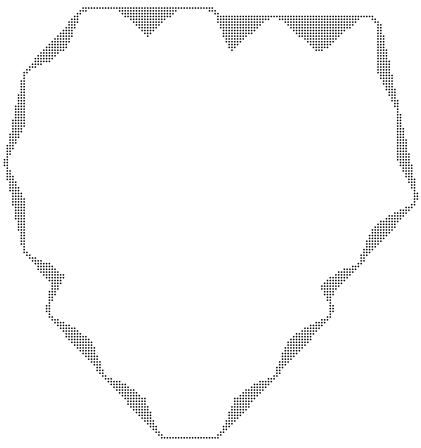
(1) Polygons



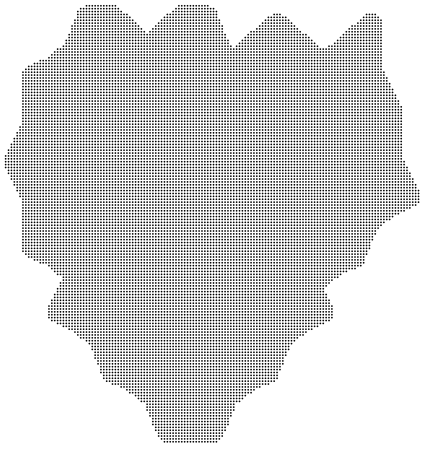
(2) S_N



(3) CNF_N



(4) S_R



(5) CNF_R

Figure 4.6: Search space construction

Algorithm 9 Incremental Search Space for Polygon in Normal Orientation

Input: A search space S_N and a cumulative no-fit space CNF_N from the previous nesting step (which can be empty), the fit spaces $F_{N\text{-}N}$, $F_{R\text{-}N}$ and no-fit spaces $NF_{N\text{-}N}$, $NF_{R\text{-}N}$ for the type of polygon being used and the position p of a polygon to be added.

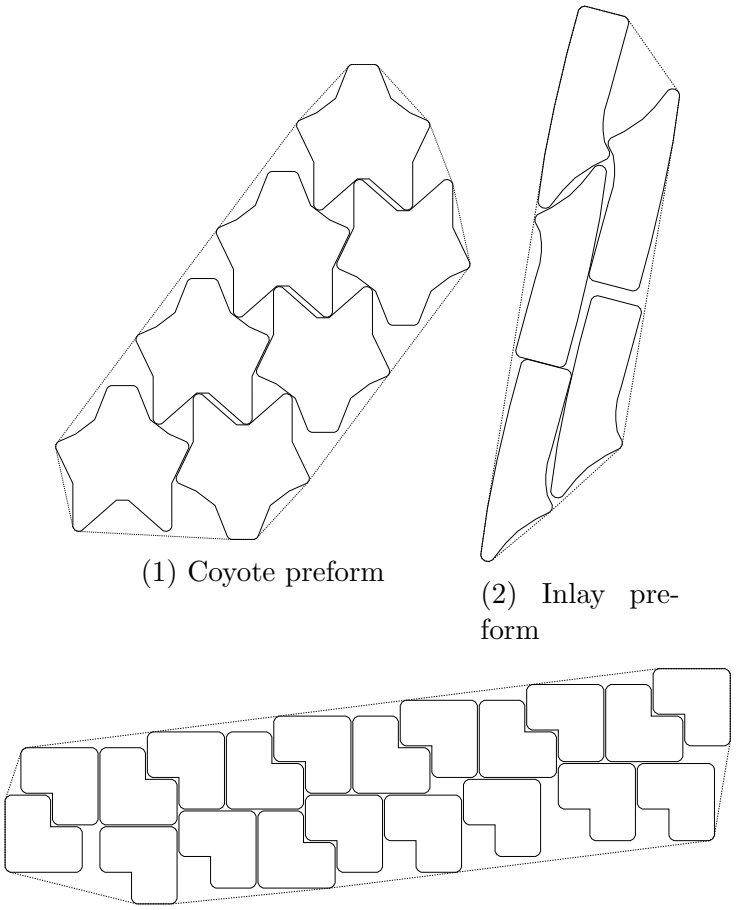
Output: An updated search space $S_N{}'$ of positions at which a polygon in normal orientation can be placed, an updated cumulative no-fit space $CNF_N{}'$.

1: If p is the position of a normally oriented polygon, let F^* be the fit space $F_{N\text{-}N}$ translated to p and NF^* be the no-fit space $NF_{N\text{-}N}$ translated to p.
2: If p is the position of a rotated polygon, let F^* be the fit space $F_{R\text{-}N}$ translated to p and NF^* be the no-fit space $NF_{R\text{-}N}$ translated to p.
3: $S_N{}' \leftarrow (S_N \setminus NF^*) \cup (F^* \setminus CNF_N)$.
4: $CNF_N{}' \leftarrow CNF_N \cup NF^*$.

4.4 Nesting Criteria

While the quality of the end result of a nesting algorithm is easy too evaluate (it's simply the number of polygons it succeeds in fitting in the given area), it's less clear how to evaluate each individual placement. Intuitively, a good placement is one in which the polygon is placed as closely as possible to the polygons that are already there or one in which as little space as possible is wasted. The question is how to operationalize these somewhat vague concepts. What is required is a metric that can be evaluated numerically in order to minimize or maximize it. We cannot take the total area covered by just the polygons themselves because that is constant for any placement. What we want is a measure for the area that is taken up by the total arrangement in the sense that it isn't available for further placements any more. This is still a somewhat vague concept, but it can be argued that the area of the convex hull polygon of the total arrangement provides at least a rough measure of it (see figure 4.7 for some examples).

The objective of a nesting criterion is to evaluate the positions in a given search space. In order to evaluate a position using the minimal convex hull area as a criterion, we place a polygon with the given orientation (depending on the search space that is currently evaluated) at that position and calculate the area of the convex hull polygon of the total arrangement of polygons. When all positions have been evaluated, we choose the position for which this area is minimal. For efficiency, we proceed incrementally, i.e. we keep the convex hull of the last nesting step, which is updated only when an actual new placement occurs. Also, for evaluation, we don't have to add every



(1) Coyote preform

(2) Inlay preform

(3) L-shape preform

Figure 4.7: Convex hull criterion

vertex of the polygon that is placed prospectively, it's enough to add its convex hull points. For calculating the convex hull efficiently, it's important to ensure that we don't have to re-sort the points by x-coordinate for each evaluation. Therefore, we store the total convex hull points in a binary search tree ordered by the x-coordinate. When we evaluate a position, we insert the translated convex hull points of the new polygon into the search tree, which is $O(m \log n)$ for m convex hull points of the polygon and n points in the total convex hull. Since the order by x-coordinate is preserved, the total convex hull can then be recalculated in $O(m + n)$, i.e. linear time. Algorithms 10 and 11 detail the steps of updating the criterion with a new placement[6] and

[6]Since we are using the contour polygon for calculating the convex hull, the last step in algorithm 10, which requires $O(n \log n)$, could be omitted, if—instead of returning the unordered list of convex hull points—we interlace the points of the upper and lower convex

the steps of evaluating a position in the search space.

Algorithm 10 Adding a polygon to the convex hull criterion

Input: A binary search tree CHT ordered by x-coordinate containing the convex hull points total from the last nesting step, a position p at which to add a new polygon, a list CHP of the convex hull points of the polygon to be added.

Output: A binary search tree CHT' ordered by x-coordinate containing the new convex hull points total.

1: $CHP^* \leftarrow CHP$ translated to p.
2: ${CHT_{list}}' \leftarrow$ convex hull of $CHT \cup CHP^*$.
3: $CHT' \leftarrow {CHT_{list}}'$ converted back to binary search tree ordered by x-coordinate.

Algorithm 11 Evaluating a position with the convex hull criterion

Input: A binary search tree CHT ordered by x-coordinate containing the convex hull points total from the last nesting step, a position p to evaluate, a list CHP of the convex hull points of the polygon to be evaluated at that position.

Output: The area of the resulting convex hull polygon.

1: Get clone CHT' of CHT.
2: $CHP^* \leftarrow CHP$ translated to p.
3: Calculate area of convex hull polygon of $CHT' \cup CHP^*$.

In tests, the convex hull criterion performs very well. One interesting feature of the convex hull criterion, though, is that it doesn't in general stress compactness of the arrangement. In particular, the criterion tends to stack polygons in a certain dimension and only proceeds to a more two-dimensional arrangement when the boundary of the nesting area is reached. It also sometimes misses opportunities for a more advantageous arrangement, e.g. in the case of the L-preform.

Since we have used the convex hull with some success, it seemed natural to test out the other two minimal bounding figures we have computed so far, i.e. the minimum bounding box and the smallest enclosing circle. Results for the minimum bounding box were highly erratic, probably because there are often different placements with more or less identical minimum bounding boxes. The smallest enclosing circle, however, is very successful in finding

hull, which can be done in $O(n)$. But since algorithm 10 contributes very little to the total run-time, so far we haven't bothered with it.

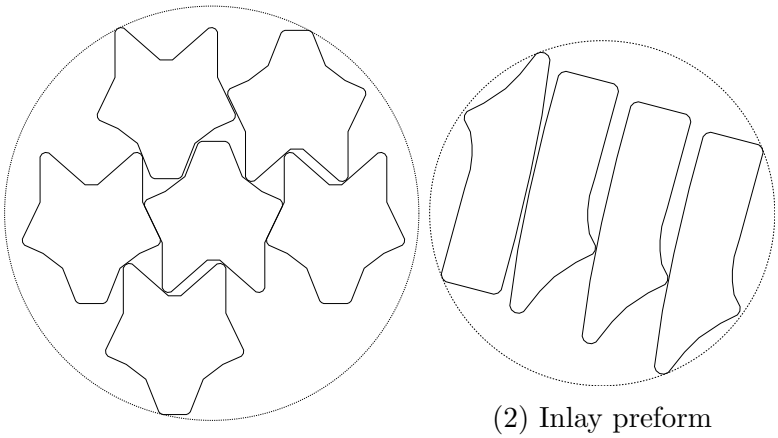
compact arrangements, especially for small numbers of polygons, e.g. pairs or quadruples (see figure 4.8 for examples). The results become less convincing when the number of polygons gets larger. Compared to the area of the convex hull polygon, the area of the smallest enclosing circle is, of course, a much coarser-grained measure of the area taken up by the polygons. The implementation of the smallest enclosing circle criterion is not much different from the convex hull criterion. We keep a list of the convex hull points of the arrangement of preforms from the last step[7] which is updated when a new polygon is placed. For evaluation, we append the translated vertices of the convex hull polygon and calculate the area of the smallest enclosing circle.

4.5 Results

For assessing and comparing the performance of the packing and nesting algorithms developed so far, I have curated a list of test cases. Any such list must be somewhat arbitrary; in this case it is a combination of basic geometric shapes, specific test cases for the algorithm and cases that are likely to be relevant for the real-world application. Figure 4.9 shows this "zoo" of preforms. Table 4.1 shows the number of preforms that were placed by box packing, circle packing and irregular shape nesting using two different nesting criteria.

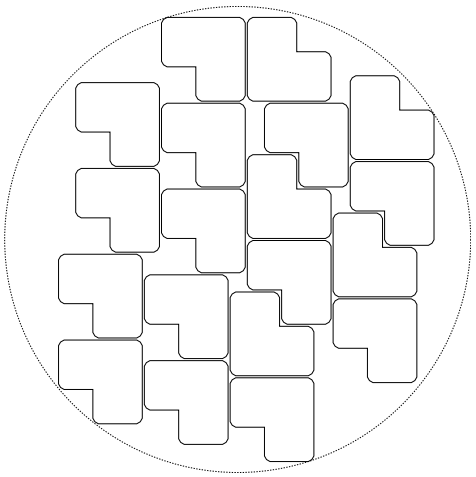
As expected, the packing algorithms do better for the cases where the bounding geometry used for packing is congruent with the shape of the preform outline, i.e. with squares and circles. In most other cases, the irregular shape nesting algorithm clearly outperforms the packing algorithms (which serve as our base line for comparison). There are a few cases where packing unexpectedly does better than nesting, for example in the case of the Letter-L and No-1 preforms. It's clear that that in those two cases, the nesting algorithm didn't succeed in finding the most compact arrangement. The performance in the case of the puzzle piece is also disappointing, because it is specifically designed to test irregular shape nesting. As for the comparison between the two irregular shape nesting criteria, it's clear that in general the convex hull criterion does much better, but there are also two cases, not coincidentaly the problematic ones (Letter L and No 1), where the smallest enclosing circle criterion does somewhat better. This is due to the relative compactness of its solutions.

[7] We don't need to keep a list of all polygon vertices because the smallest enclosing circle of a polygon and that of its convex hull are identical. It's not necessary to keep the points sorted by x-coordinate since the randomized incremental construction algorithm for the smallest enclosing circle randomizes the set.



(1) Coyote preform

(2) Inlay preform



(3) L-shape preform

Figure 4.8: Smallest enclosing circle criterion

One general problem which makes the comparison somewhat misleading is that in the case of irregular shape nesting, so far there has been no real attempt to fit the arrangement of preforms to the circular nesting area. We have simply started with a central placement and then proceeded until no more legal placements were available. The packing algorithms, on the other hand, each had a strategy for fitting the packing to the boundary. In box packing we used a variable start position of the rows or columns and also calculated the length available in each row or column. This is one reason why box packing does so well with more-or-less rectangular shapes like Letter L and No 1. In circle packing we generated a larger packing and then tried to find a position for the circular boundary that contains the most poly-

Preform	Packing		Irregular Shape Nesting	
	Box Packing	Circle Packing	Convex Hull Criterion	Smallest Enclosing Circle Criterion
Square	37	22	32	31
Circle	80	90	85	83
Triangle	24	16	38	35
Ellipse	163	90	167	158
Letter L	80	48	74	77
No 1	80	48	78	78
Angle	80	60	104	104
Dog Bone	28	19	33	29
Puzzle Piece	28	22	28	27
Coyote	14	12	19	16
Inlay	33	10	36	31

Table 4.1: Comparison of packing and nesting results

gons. This second strategy could also be used for irregular shape nesting by generating a larger nesting and then finding the best clipping of it.

Apart from realizing a significant increase in the number of polygons placed in many cases, which shows that the approach is basically sound, the nesting algorithm developed in this chapter also proves its computational feasibility (see table 4.2 for typical run times). At least in the case of the convex hull criterion, computing a complete nesting can be done in under a second. The smallest-enclosing-circle criterion is much slower, despite the fact that due to its expected-linear time it should be roughly on equal footing. But as its results show, it is probably not suited as a general nesting criterion anyway. The run times for the convex hull criterion is affected mostly by the number of vertices in the offset curve polygon. The fastest times are for square and triangle, which have 12 and 11 vertices, respectively, in their offset curve polygons; the slowest are circle and ellipse, which have 72 and 93 vertices. Another variation occurs for convex vs. non-convex shapes. For example, the Letter L preform has 16 vertices, so only a few more than the square, but over twice the run time, which is likely due to the fact that the fit spaces for non-convex shapes are not as parsimonious as the ones for convex shapes.

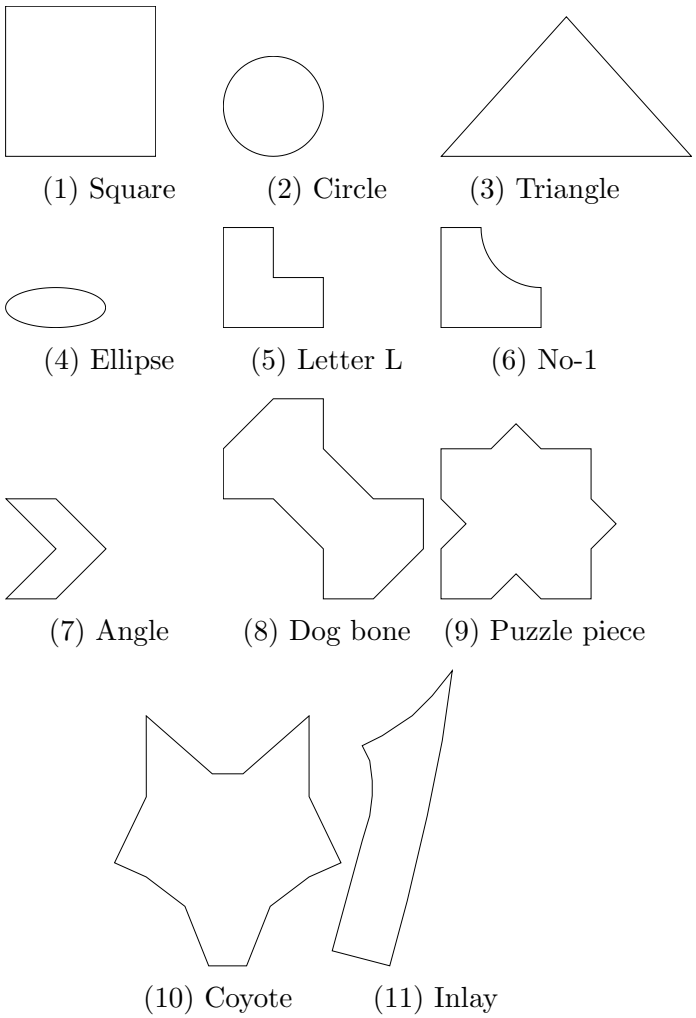


Figure 4.9: Preform test cases

Preform	Packing		Irregular Shape Nesting	
	Box Packing	Circle Packing	Convex Hull Criterion	Smallest Enclosing Circle Criterion
Square	6 ms	55 ms	165 ms	3342 ms
Circle	1 ms	145 ms	1717 ms	88 541 ms
Triangle	1 ms	45 ms	190 ms	5178 ms
Ellipse	3 ms	134 ms	3088 ms	210 824 ms
Letter L	1 ms	95 ms	369 ms	23 247 ms
No 1	2 ms	90 ms	511 ms	24 401 ms
Angle	1 ms	71 ms	419 ms	34 647 ms
Dog Bone	2 ms	51 ms	461 ms	13 718 ms
Puzzle Piece	1 ms	45 ms	343 ms	10 007 ms
Coyote	1 ms	49 ms	481 ms	10 463 ms
Inlay	0 ms	48 ms	356 ms	9004 ms
Average	1.7 ms	75.3 ms	736.4 ms	39 397.5 ms

Table 4.2: Packing and nesting run time comparison

Chapter 5

Tuple Nesting

While we were able to demonstrate the general viability of our approach to irregular shape nesting with the one-by-one nesting strategy, there still is room for improvement in some areas. For one, there are several preforms for which the results aren't satisfactory with respect to nesting density. For another, we still don't have a general solution for the problem of finding the best fit to the boundary of the nesting area. The "tuple nesting" strategy presented in this chapter tries to address these problems.

In the context of evaluating the convex hull and the smallest enclosing circle with respect to their suitability as a criterion for the quality of a placement, it emerged that the convex hull is well-suited as a kind of all-purpose criterion; it works well for a wide range of preforms and it also works well for larger arrangements of preforms. The smallest enclosing circle, however, in some cases provides a superior nesting quality for small tuples of preforms. This suggest the following hybrid strategy: using the smallest enclosing circle criterion for creating dense tuples of preforms (pairs, quadruples) and then using the convex hull criterion to nest these tuples. This would also facilitate adapting a strategy developed in the case of circle packing for solving the problem of fitting the nesting to the boundary of the nesting area, namely using tuple nesting to generate a kind of plane packing and executing a search pattern over this packing.

5.1 Tuple Generation

In principle, we could use one-by-one nesting to generate tuples of arbitrary arity. But since we want to use tuples to cover (a finite portion of) the plane, we would prefer an approach that promotes some regularity of the resulting tuples and also gives us a neat combinatorics for choosing between different

kinds of tuples. For these reasons, it seemed more natural to start by constructing pairs from individual polygons and then constructing quadruples from these pairs. This gives us the following combinatorial possibilities:

- There are two possible individual polygons: *N* (normal orientation), *R* (rotated by 180°).
- There are three possible pairs of polygons: *NN* (both normally oriented), *NR* (one normally oriented, one rotated), *RR* (*NN* rotated).
- There are three salient quadruples: *NNNN* (two pairs *NN*), *NRNR* (two pairs *NR*), *NNRR* (one pair *NN*, one pair *RR*).

While the three pairs exhaust the possibilities of combining polygons, we have been more selective for the quadruples: In theory, it would, of course, be possible to construct a quadruple *NN-NR*, for example, but it's unlikely to result in a nice, regular covering of the plane.

We can use one-by-one nesting only for the first step of generating pairs from individual polygons. The search spaces required for nesting pairs into quadruples cannot be generated the same way we generate the search spaces for individual polygons. It would be possible, of course, to calculate no-fit and fit spaces for pairs from scratch, but the more elegant solution is to generate them from the fit and no-fit spaces of the individual polygons, which contain all the necessary information.

The basic idea of how to construct a search space for, e.g. nesting a pair *NN* next to another pair *NN* is this: We can construct the search space *NN-N* (i.e. the search space for nesting a polygon *N* next to to a fixed pair of polygons *NN*) in the way outlined in chapter 4 by constructing it from the fit and no-fit spaces *N-N*. This search space *NN-N* gives us the set of vectors at which we might place a polygon *N* next to the pair *NN*. Conversely, the set of negative vectors to these vectors give us the positions at which we might place the pair *NN* next to a polygon *N*. So in order to construct the search space *N-NN*, we only need to construct *NN-N* and then take the negative vector of each vector.[1] With this space *N-NN*, we can then construct *NN-NN* again by the kind of search space construction introduced in the last chapter. The same procedure can be repeated to construct the search spaces for nesting quadruples: We use *NN-NN* to construct *NNNN-NN*, which gives us *NN-NNNN*, from which we construct *NNNN-NNNN*.

In total, the tuple generation step of tuple nesting involves the following: We first use the basic fit and no-fit spaces to create pairs *NN*, *NR* and

[1] If we take the vectors as position vectors, this operation is equivalent to point reflection around the origin.

RR. We construct the search spaces *NN-NN*, *NN-RR* and *NR-NR* for these pairs and use them to generate the quadruples *NNNN*, *NNRR* and *NRNR*. Finally, we construct the search spaces *NNNN-NNNN*, *NNRR-NNRR* and *NRNR-NRNR* for nesting those quadruples.

Evaluating these search spaces isn't much different conceptually than evaluating the search spaces in chapter 4. We simply have to generalize the concept of a nesting unit, which can now be a single polygon or a polygon set. For each of the three steps, i.e. nesting singles into pairs, nesting pairs into quadruples, nesting quadruple into a plane nesting, we have the option of applying a different nesting criterion. In practice, the convex hull criterion is always the best choice for the third step, but for the first two steps, there are different combinations of criteria which perform best depending on the particular geometry of the preform.

5.2 Plane Nesting and Search

In order to do better on the issue of fitting the nesting to the boundary, we want to use the three different quadruples generated in the first two steps of the tuples strategy to generate larger nestings corresponding to the plane packings in chapter 3 in order to be able to do a meaningful search on them. Ideally, we would be able to determine periods of the nesting (i.e. translations along some axis that preserve the shape of the pattern). We could then generate a pattern that allows for a search along these axes of translational symmetry with a range corresponding to these periods. As a rough substitute, we stipulate that the search should have a range corresponding to the dimensions of the smallest enclosing circle of the quadruple (which in most cases is the most "generous" of the three types of minimal bounding geometries) that was used to generate the nesting. This results in a circular nesting area with a radius equivalent to the sum of the table radius and the radius of the smallest enclosing circle of the quadruple. Since we are nesting not single polygons, but polygon sets, the nesting algorithm terminates when no more quadruples can be placed. We then generate a search space with a given rasterization in the shape of the smallest enclosing circle of the quadruple. For each position in this search space, we place a circle representing the table edge centered on the search position and filter out the set of offset curve polygons contained by the circle. Each of the three nestings based on the tuples *NNNN*, *NNRR* and *NRNR*, respectively, yields a result; we choose the maximal of these results.

5.3 Results

In order to assess the results of the tuple nesting algorithm, we have tried out different combinations of nesting criteria. As pointed our earlier, we always use the convex hull criterion for the final step because the results of the smallest enclosing circle criterion are unsatisfactory for larger arrangements. For the first two steps, i.e. for nesting single polygons into pairs and for nesting pairs into quadruples, the most successful combinations were using the convex hull criterion for both steps (type 1), or using the smallest enclosing circle criterion for both steps (type 2). See table 5.1 for a comparison between the best irregular shape nesting results and the results for type 1 and type 2 tuple nesting. The first observation is that tuple nesting manages to improve upon irregular shape nesting in every case. The second observation is that, as expected, type 2 tuple nesting (smallest enclosing circle) works slightly better overall and especially shines for preforms for which the compactness of the arrangement is an asset (e.g. Letter L and No 1), while type 1 tuples nesting (convex hull) works better for polygons for which compactness is rather an empediment, e.g. the inlay preform. Table 5.2 compares the best tuple nesting results with the best packing results, which served as our baseline. We now see that tuple nesting outperforms packing in every instance except for the two basic shapes (square and circle), even in the cases of Letter L, No 1 and the puzzle piece, which were found lacking in chapter 4. One reason it even comes close for the square and the circle is that as an unintended consequence of our nesting criteria (which are both based on convex shapes), the shape and arrangement of tuples conforms rather well to the circular area. Figure 5.1 shows some of the "winning" arrangements. Both density and fitting to the boundary are now excellent throughout. The slight underperformance in the case of the square and the circle can easily be solved by running box and circle packing, whose run times are negligible, in addition to tuple nesting. The only two cases where there might be room for some marginal improvement are the triangle and the angle preform. The reason for this is that even after nesting single polygons into tuples, the tuples retain a degree of freedom. Essentially, triangles or angles are combined into rows and it would then be possible to fit these rows to the width of the nesting area as we did in box packing in chapter 2. It could be a direction for further research to develop this hybrid strategy.

Table 5.3 show typical run times for the two types of tuple nesting. Since a nesting strategy that is maximal for all performs would have to run both types for an average total run time of 17.7 s, these run times are not entirely satisfactory given our goal of being done after < 1 s. But the fact that we are only roughly one order of magnitude away makes it very likely that the run

Preform	Irregular Shape Nesting	Tuple Nesting	
		Type 1	Type 2
Square	32	34	34
Circle	85	88	83
Triangle	38	40	41
Ellipse	167	168	171
Letter L	77	77	87
No 1	78	81	89
Angle	104	108	109
Dog Bone	33	34	33
Puzzle Piece	28	32	32
Coyote	19	21	20
Inlay	36	39	33

Table 5.1: Comparison of tuple nesting results for different types

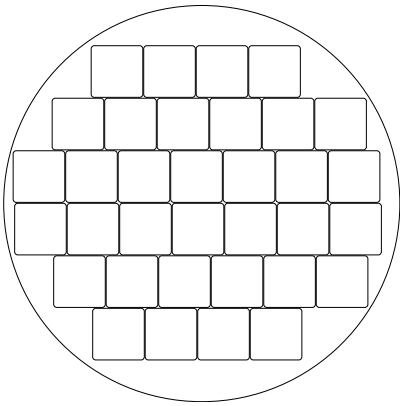
time could be brought down to this goal with some rounds of refactoring and optimization. Since the tests were run on a very modest Intel Pentium G3220 @ 3.00GHz with two cores, even just running it on a current mid-range CPU might do the trick. Since the bulk of the run time is spent evaluating search spaces, which could be done in parallel, one could use parallel streams from Java's Stream API to run the evaluation on multiple cores. There are also plenty of opportunities for further optimizing the code, some of which I have indicated in past chapters. Search space operations could be accelerated by encoding integer vectors in integer numbers. Although both convex hull and smallest enclosing circle have optimal-time implementations there is probably room for improving the constant. One could also iterate the generation of tuples, i.e. nest quadruples into 8-tuples, 16-tuples, etc., which would bring down the number of nesting steps required in the final step exponentially.

Preform	Packing	Tuple Nesting	Ratio
Square	37	34	92 %
Circle	90	88	98 %
Triangle	24	41	171 %
Ellipse	163	171	105 %
Letter L	80	87	109 %
No 1	80	89	111 %
Angle	80	109	136 %
Dog Bone	28	34	121 %
Puzzle Piece	28	32	114 %
Coyote	14	21	150 %
Inlay	33	39	118 %

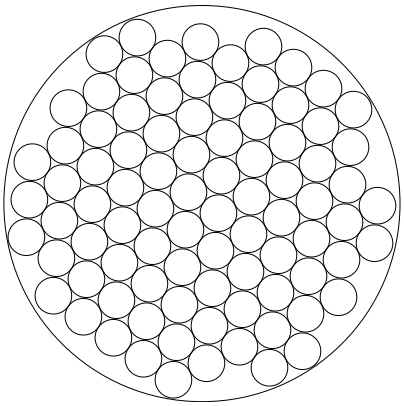
Table 5.2: Comparison of packing with tuple nesting

Preform	Tuple Nesting	
	Type 1	Type 2
Square	5345 ms	2971 ms
Square	4894 ms	3354 ms
Circle	15 718 ms	12 012 ms
Triangle	3644 ms	3650 ms
Ellipse	31 253 ms	20 901 ms
Letter L	5838 ms	3759 ms
No 1	10 397 ms	6192 ms
Angle	5202 ms	3525 ms
Dog Bone	7002 ms	5141 ms
Puzzle Piece	5603 ms	5812 ms
Coyote	10 045 ms	7297 ms
Inlay	17 657 ms	5172 ms
Average	10 659 ms	6983 ms

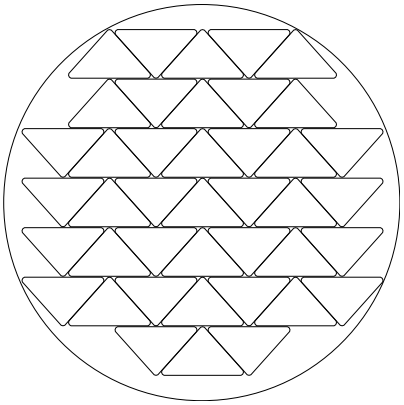
Table 5.3: Run times for tuple nesting



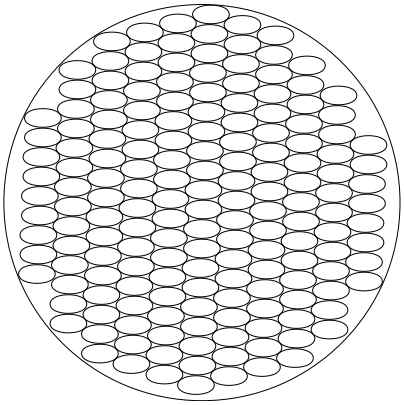
(1) Square



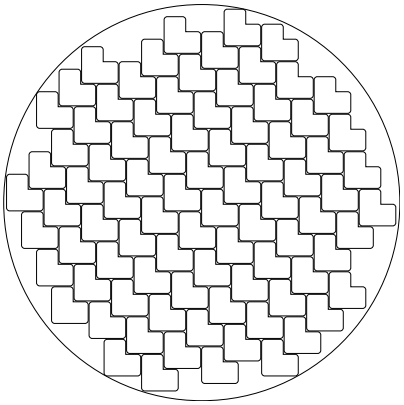
(2) Circle



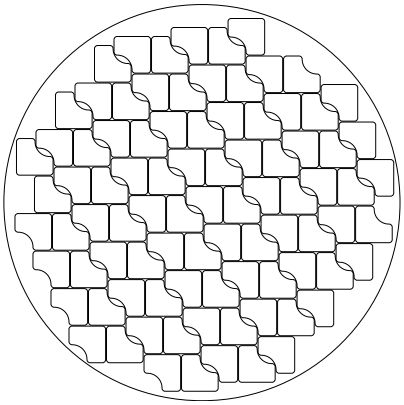
(3) Triangle



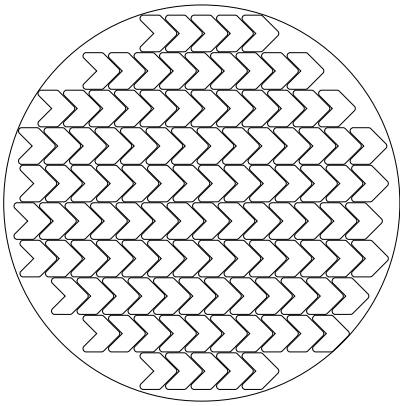
(4) Ellipse



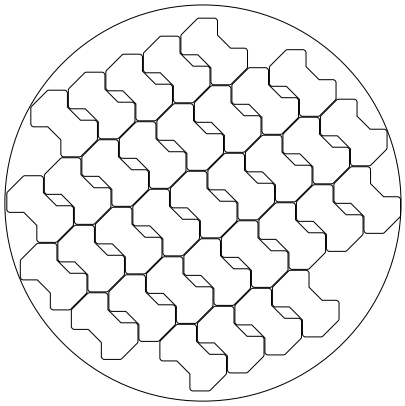
(5) Letter L



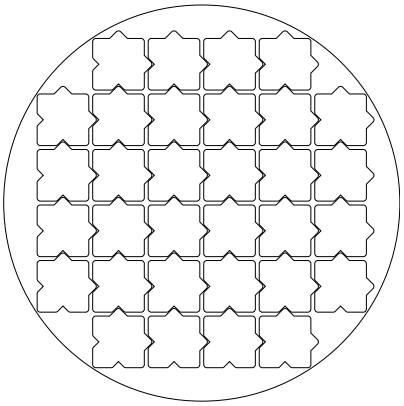
(6) No-1



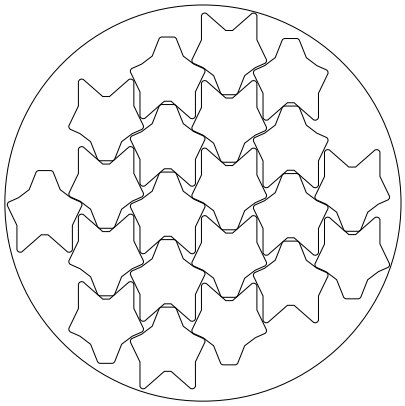
(7) Angle



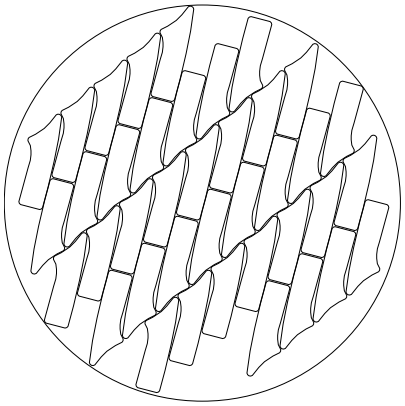
(8) Dog bone



(9) Puzzle piece



(10) Coyote



(11) Inlay

Figure 5.1: Maximal tuple nesting results

Bibliography

[1] Julia A Bennell and Jose F Oliveira. "The geometry of nesting problems: A tutorial". In: *European journal of operational research* 184.2 (2008), pp. 397–415.

[2] Mark de Berg et al. *Computational Geometry. Algorithms and Applications*. 3rd ed. Berlin: Springer-Verlag, 2008.

[3] *Circles in Circles*. Nov. 3, 2021. URL: `https://erich-friedman.github.io/packing/cirincir/`.

[4] *Convex hull algorithm*. Sept. 24, 2021. URL: `https://www.nayuki.io/page/convex-hull-algorithm`.

[5] Ray Cuninghame-Green. "Geometry, shoemaking and the milk tray problem". In: *New Scientist* 123.1677 (1989), pp. 50–53.

[6] J.-M. Drappier. *Enveloppes convexes*. Tech. rep. Centre CPAO, ENSTA Palaiseau, 1983.

[7] H. Freeman and R. Shapira. "Determining the minimum-area encasing rectangle for an arbitrary closed curve". In: *Communications of the ACM* 18.7 (1975), pp. 409–413.

[8] Pijush K. Ghosh. "An algebra of polygons through the notion of negative shapes". In: *CVGIP Image Underst.* 54 (1991), pp. 119–144.

[9] Rolf Klein. *Algorithmische Geometrie: Grundlagen, Methoden, Anwendungen*. Berlin: Springer-Verlag, 2006.

[10] Jonas Lindmark. *No Fit Polygon Problem*. 2013.

[11] Hormoz Pirzadeh. "Computational geometry with the rotating calipers". McGill University, 1999.

[12] Michael Ian Shamos. "Computational geometry". Yale University, 1978.

[13] Sven Skyum. "A simple algorithm for computing the smallest enclosing circle". In: *Information Processing Letters* 37.3 (1991), pp. 121–125.

[14] *Squares in Circles*. Sept. 27, 2021. URL: https://erich-friedman.github.io/packing/squincir/.

[15] *The best known packings of equal circles in a circle (complete up to N = 2600)*. Nov. 3, 2021. URL: http://hydra.nat.uni-magdeburg.de/packing/cci/cci.html.

[16] Godfried T Toussaint. "Solving geometric problems with the rotating calipers". In: *Proceedings of IEEE MELECON*. Vol. 83. 1983, A10.

[17] Emo Welzl. "Smallest enclosing disks (balls and ellipsoids)". In: *New results and new trends in computer science*. Springer, 1991, pp. 359–370.

[18] Wikipedia contributors. *Parallel curve — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Parallel_curve&oldid=1073067640. [Online; accessed 22-February-2022]. 2022.

Erklärung

Ich erkläre, dass ich die Masterarbeit selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für enthaltene Zeichnungen, Skizzen oder graphische Darstellungen. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Arbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate geprüft werden kann und ausschließlich für Prüfungszwecke gespeichert wird.

Berlin, 28.02.2022

_____

Markus Säbel