Università di Genova
DIBRIS DIPARTIMENTO
DI INFORMATICA, BIOINGEGNERIA,
ROBOTICA E INGEGNERIA DEI SISTEMI

Evolutionary Computation Methods for Nesting Problem with Continuous Rotation

MOMINA SAJID

DIBRIS, Università di Genova
Via Opera Pia, 13 16145 Genova, Italy
http://www.dibris.unige.it/

Master Thesis

Laurea Magistrale in Computer Science
MSc in Computer Science
Data Science and Engineering Curriculum

Evolutionary Computation Methods for Nesting Problem with Continuous Rotation

Momina Sajid

Advisor: Prof. Francesco Masulli
Prof. Stefano Rovetta
Examiner: Prof. Francesca Odone

04th October, 2023

To my father, whose unwavering support and belief in my potential have been the keystone in this achievement.

Abstract

The problem of cutting and packing of irregular pieces or Nesting problem, focus to find the finest solution of the parts within the range, with the primary goal to minimize the height used by pieces. This problem holds significant relevance in industries such as furniture, textiles and footware, where the cutting and packing of pieces are most important. The problem addressed is two-dimensional problem with convex and non-convex pieces with free rotation. Our principle key focus on to be able to answer the following research question: “Is there any way to improve the efficiency of two-dimensional nesting problem with free rotations, using evolutionary algorithms?” To address this question, a set of five sorting rules, two positioning rules, and two rotation rules were implemented. A Genetic Algorithm (GA) and Particle-Swarm-Optimizarion Algorithm (PSO) were proposed, each using combinations of a positioning rule, a sorting rule, and both rotation rules.
Computational experiments conducted with ten instances of the literature indicated, that use two positioning rules (calledαandβ). In particular, one of them obtained better results . A Particle-Swarm-Optimization Algorithm and a Genetic Algorithm were also implemented, which use the combination of these two position rules that stood out. We use DEAP (distributed evolutionary algorithms in python) for the implementation of our evolutionary algorithms. Than we perform different experiments with multiple built-in features of deap, such as different built-in operators for crossover, mutation and selection methods in Genetic alorithm. For the experiments of Particle-Swarm-Optimization algorithms different operators is used and the comparison is made in experiments section of the work.

Keywords:Nesting, Continuous Rotation, evolutionary algorithms.

Table of Contents

Chapter 1

Introduction

The problem of cutting and packing irregular parts, known as nesting, has an objective to figure out the layout for a limited amount of convex and non-convex pieces (items), wasted space on a board. In industries including furniture, paper, metals, testile and leather, decision-maker are always focused on managing productive raw materials. Therefore, to increase production profitability, such solutions heavily emphasize sustainability. In the work of [1], nesting is also reffered to as an irregular strip packing problem.
We will address a form of the nesting problem with continuous rotation in this project. A problem where we have the width to be fixed plate, and the height to be minimized (in this case, the plate can be called a strip). The parts that will be placed in the strip will have two dimensions and can be rotated freely. The range in which items have to arranged is rectangular and its fixed dimensions is known in advance.
The problem of nesting contains some restrictions that must be respected in order to obtain an optimal solution. Such restrictions are:
Containment restriction, which guarantees:

- All parts within the range.
- All parts must be allocated.
- No parts will overlap each other’s.

The way the pieces are represented, how the overlap between them is verified, and confinement to the strip form the fundamental selection measures to assess an algorithm's performance.
In particular, several authors have geometric approaches that account for the overlap of items, such as raster methods, direct trigonometry, no-fit polygons [2], no-fit raster [3], and separate lines [4].
In the literature, several methods for resolving the cutting and packing problem have already been put forth. however, few works think about the free rotation of the parts. When considering, the optimization models the problem turns into nonlinear. A few works that take this feature of the pieces are those of [4], [5], and [6] which address the problem of nesting with continuous rotation through a model with separating lines and a formulation with separating parabolas.

In this work, we implement heuristics methods of the paper by [7] to improve the state of art of solving the uneven cutting and packing being concern parts with free rotation, to minimize the waste generated by the allocation of the parts.
In this paper, an approach using a genetic algorithm to the continuous rotation nesting problem is proposed. (i.e. finding the finest position of irregular parts in a strip, minimising the space they use). In particular, they introduce two rules for positioning, five rules for sorting and two rules for rotation of parts to solve the problem of irregular shapes.
Additionally, as part of our strategy to address the nesting problem, we intend to use both the Genetic technique (GA) and the Particle Swarm Optimization (PSO) technique.
We have carried out extensive studies to evaluate and compare the performance of the Genetic Algorithm (GA) and Particle Swarm Optimization (PSO) algorithms.
These experiments involved solving various instances of the cutting and packing problem with free rotation using both algorithms.
The results obtained from each algorithm are analyzed and compared.

1.1 Objectives

The objectives of this work are:

- Do a literature review of works that address the problem.
- Implement different heuristics to solve the problem.
- Implement different evolutionary algorithms to compare the results.

1.2 Text Structure

The rest of the work: Chapter 2 presents DEAP User-Manual tutorial. In Chapter 3, we presented the methods proposed in paper for solving the nesting problem with free rotation and some methods for detecting the overlap between parts. Than Chapter 4 contains the algorithms that were implemented for solving the problem. In Chapter 5, the Computational results of the algorithms are presented for the challenge of nesting with free rotation. From the results obtained, we present in Chapter 6, our final considerations of the work.

1.3 Problem Definition

In simpler terms, consider a rectangular sheet of material. The sheet's width (measured along the x-axis) is set but its height (measured along the y-axis) can fluctuate. The goal is to cut out numerous odd shapes from this sheet while keeping the total height of the sheet to a minimum. We can also rotate the shapes in any direction to make them fit better. The goal is to determine the optimum way to arrange and rotate these forms in order to use the least amount of material.

Assume that the bottom-left corner of our rectangular sheet begins at the grid point (0,0). We now have a collection of forms, each of which is represented by a polygon. These polygons are composed of a succession of points or “vertices”, each of which has its own (x, y) location on the grid.
We can rotate these polygons to make them fit better on the page. When we rotate a polygon, we are spinning all of its vertices by the same angle around a point. The goal is to discover the best arrangement and rotation of these forms in order to reduce the required sheet height.
In this NPCR (Nesting Problem with Continuous Rotations) work, we must fit each of our polygons into the rectangular sheet in a precise way.

There are two major rules to follow:

- Every point on every polygon must fit within the sheet's width. In other words, the x-coordinate of each vertex must be between 0 and L, where L is the sheet's fixed width.
- The polygons cannot be overlapping. So, if we have two polygons p and $'$ p $'$, they must be arranged so that they do not collide or lay on top of one another.

The goal is to arrange and rotate these polygons such that they all fit within the sheet without breaking any of the restrictions and while keeping the total height of the sheet to a minimum.

1.4 Literature Review

This work involves two main topics, we deal with how parts are represented in the literature and how author’s deal with the overlapping parts. Then we deal with the works that consider the free rotation of the irregular pieces. Then we also present some works in the literature that focus on the nesting problem without free rotation.

1.5 Representation of parts and how to avoid overlapping between them

In the work [8] they proposed a representation scheme of parts used generating arrays with values 0’s and 1’s. when a matrix cell contains the value 1, it means that there is a piece in the region. With this type of representation, analyzing whether two parts are overlapping becomes an easier task. When two pieces overlap, the value of their intersection cells becomes greater than 1. Such a representation of pieces can be seen in (figure 1.1).

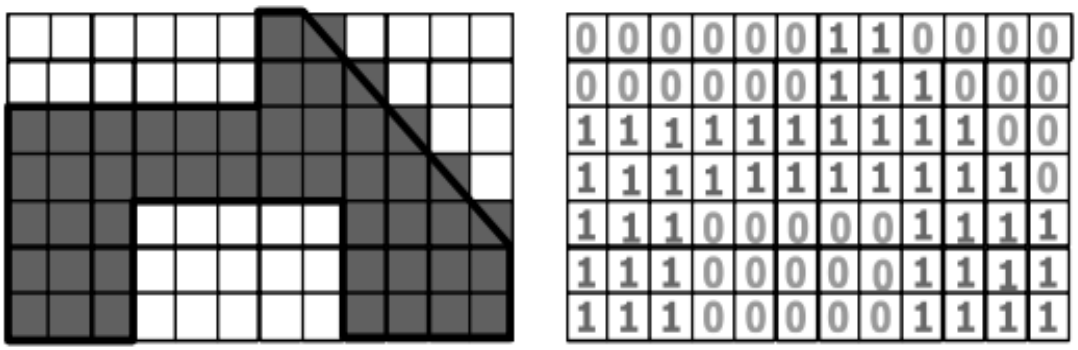

Figure 1.1 Representation of parts in matrices

Another representation used consist of transforming each piece of the problem into a polygon. Each vertex of the piece is consider by a point (x, y) in the cartesian plane, thus transforming the piece into a polygon. For such a representation, the vertices are stored ordered in a data structure [4]. With the modeling of the parts through polygons, the parts can be better represented and the use of computational memory is smaller than the representation using matrices. However, verifying their non-overlapping becomes a more difficult task.

In the work by [2] some methods for verifying overlapping parts are presented and the computational difficulty of each method is shown. For the case in which the parts are represented by polygons, a method presented is called trigonometry, which consist of working directly with the vertices and border of the polygons to verify the non-overlapping of the polygons. One possible way to authenticate this claim is by conducting the subsequent examination:

- Test 1: Do the polygon's overlap?
  No – Polygons do not overlap. (Figure 1.2, a).
  Yes – Apply Test 2.

- Test 2: For every set of two edges originating from distinct polygons, is there an overlap in their corresponding bound boxes?
  Not for all – polygons do not overlap. (Figure 1.2, b).
  Yes – For each edge apply Test 3.

- Test 3: For each set of two edges of the different polygons, is there any intersection?
  Not for all – Apply Test 4. (Figure 1.2, c).
  Yes – Polygons Overlap.

- Test 4: For each vertex of polygons, is it within the other polygon?

No – There is no overlapping of polygons. (Figure 1.2, d).
Yes – Polygons overlap. (Figure 1.2, e).

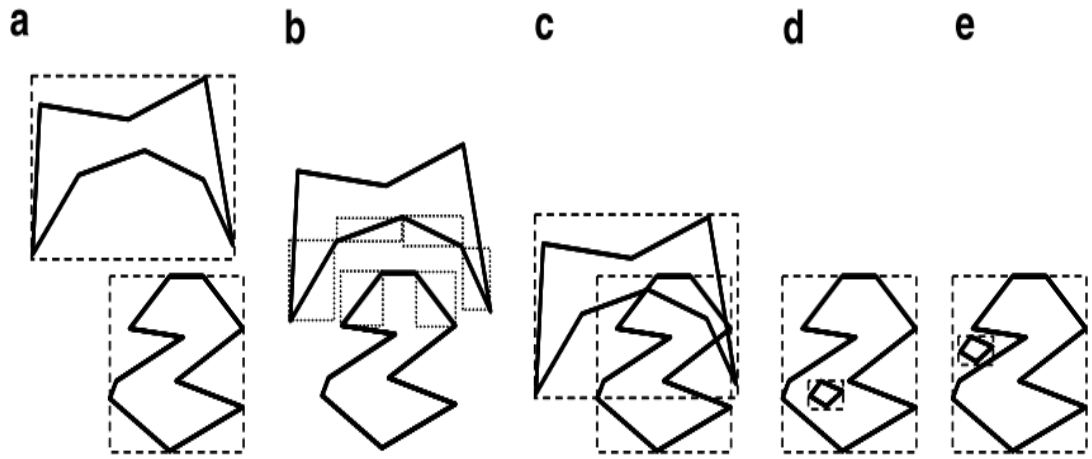

Figure 1.2 Some positions for two polygons

The most used methods in the literature for checking overlapping of parts, when these are represented by polygons, is the No-Fit polygon (NFP). The fundamental of the NFP is a polygon obtain from the mixture of two polygons, where its innermost states the points of overlap between the items (Figure 1.3). After building the NFP between two polygons, checking for non-overlapping between them comes down to checking whether a point is inside or outside the NFP. The construction of NFP is not a simple task, however it can be accomplished during a pre-processing stage, enabling the efficient execution of the non-overlap verification [2].

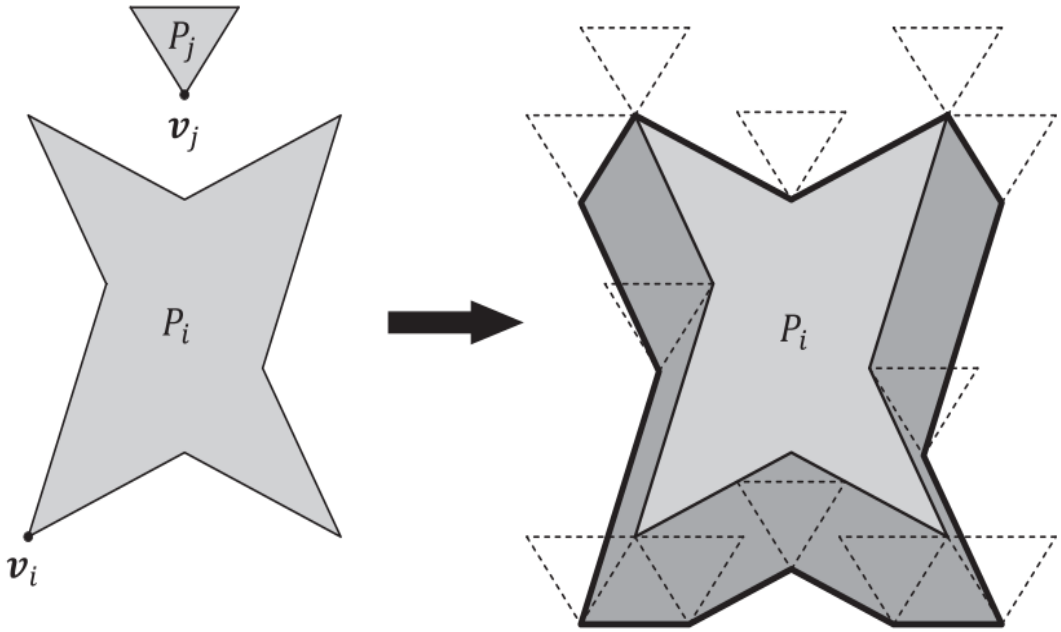

Figure 1.3 Representation of a no-fit Polygon

The work [9] Introduce a non-overlapping verification approach, known as No-Fit-Raster method. Such an algorithm has similarities with the standard No-Fit Polygon. The No-Fit Raster represents the NFPAB (No-Fit Polygon between part A and B) as a Boolean mesh, with resolution g. If g= 1, then the plane will be discretized into integers. Within the mesh the value of each position is 1 if it lies.
Within the polygon generated by NFP, or 0 if it resides outside the polygon (Figure 1.4).

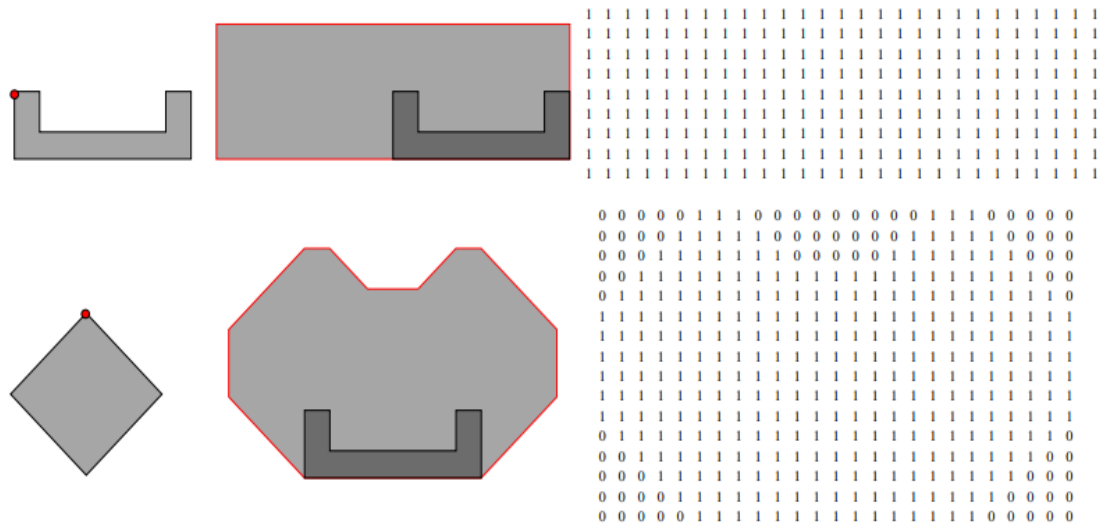

Figure 1.4 no-fit-polygon and its no-fit-raster

The work [10] has an alternative representation which is less commonly utilized involves representing items through the use of circles and ellipses for the approximation of their shapes. With this illustration, the assessment of overlap among items is computed based on the circles and ellipses this works to depict these items. As demonstrated in (Figure 1.5), the author introduces an exact technique that resolve the issue of overlap among circles. This method not only deal with overlap issue but also able to include more circles in cases where items overlap occur.

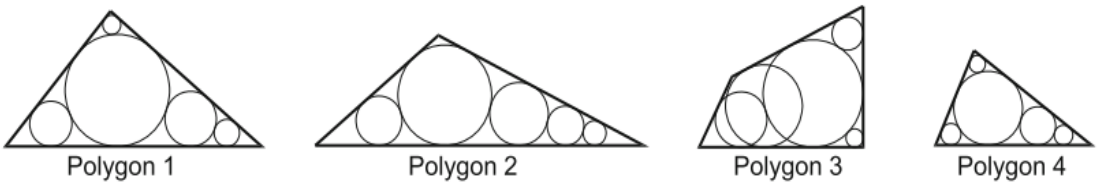

Figure 1.5 Four polygons together with their initial circles

In this work [4], A separation method using separating lines and parabolas was introduced. We can say that two convex polygons do not overlap if the point of each of them are on opposite side of the lines in question. In the situation of non-convex polygons, the pieces are separated into convex polygons and a straight line is created for each piece. Figure (1.6) depicts how the presentation of the separating lines are shown.

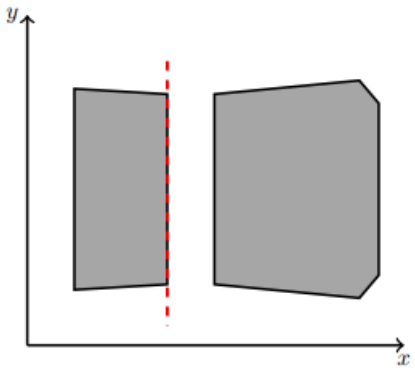

Figure 1.6 Separating line of two convex polygons

When the problem consists of packing convex pieces, the separating lines have a much better performance. However, when a piece is non-convex [4] introduced the use of separate parabolas to ensure non-overlapping. A parabola separates two pieces if one piece is within the parabola and the other is outside the parabola. When employing separator lines for the purpose of separating convex polygons, it is always possible to touch one piece to the other. However, in the case of non-convex piece, achieving such contact is not always possible. In this case, the utilization of a separator parabola becomes an option that may allow that non-convex pieces touch each other.

1.6 Nesting problem with free rotation

To our uniderstanding, there exist few studies that addresses the nesting challenge with free rotation of parts. The difficulty of combining objects of varied sizes while allowing them to spin freely is addressed in the study by [11]. They accomplish this via a two-step process. They initially concentrate on arranging the more significant objects so that there is space for the smaller ones. They employ a theory called as “Complete Coverage of Circles” to depict these things. In the subsequent stage, they use the smaller pieces to close the gaps. In every test situation, their approach outperformed conventional compression algorithms. However, it takes a little longer time to complete the task.

Based on a physical model by [5] and his team developed a packing method that resembles a rubber band in 2016. The objects that need to be sorted in their problem are shown as polygons. These polygonal objects are pulled together like rubber bands until they are tightly compacted. To do this, the algorithm separates the objects into four sections outside of the main board. Then, the distance between their vertices and mass is calculated, to detemined how compact these polygons can be. Then, like a rubber band, these elements are drawn toward the center of the board. When everything is contained within the board’s predetermined dimensions, the “rubber band” keeps getting tighter.

A mixed-integer programming model with quadratic limitations was developed in 2018 by [12]. In this paradigm, components are positioned while accounting for their rotation and location based on a reference point. It was not ideal that the model initially produced so many symmetric solutions. In order to address this, they added symmetry-breaking constraints, which, raised the caliber of the solutions the model generated. The model was examined using three different solvers: COUNNE, SCIP, and BARON. They used more than one technique to solve the problem. Among them, SCIP emerged as the standout performer, producing the best outcomes.

The author in [4] Created a novel method of item sorting in 2017. To increase the efficiency of the sorting, “separator parabolas,” which are lines and unique curves, were developed. They took it a step further by developing a sophisticated mathematical model to support their approach. They altered the “Bottom-Left algorithm,” an existing technique, to evaluate its performance. They used a different strategy known as the “point algorithm interiors” to further refine their findings. They were essentially looking for a more effective solution to the sorting issue they were concentrating on.

In order to prevent objects from overlapping when we are trying to fit them together, [6] created a new method in 2016. They accomplish this via a unique function φ. They started by developing an algorithm that identifies adequate solutions quickly. This approach begins by locating smaller areas where objects can fit without overlapping with certainty. Because the algorithm starts in these “safe zones,” it finds a complete, functional answer more quickly. This is due to the fact that as time passes, fewer constraints become an issue. They discovered that their method outperformed four other ways from earlier studies when they put them to the test.

There are some other works that deal with nesting problem without free rotation. In the work A specific method known as "guided cuckoo search" was employed in 2013 study by [13] to address the shape-fitting or nesting problem. Shapes that naturally fit into gaps between one another are first grouped together by the algorithm. After sorting them, it creates a preliminary answer via a different technique known as "Bottom-Left." The guided cuckoo search then tries to make better use of the area that is available. Some shapes throughout this process might overlap, therefore the algorithm moves them about until they all fit together without doing so. The technique demonstrated 100% efficiency in tests, and it now has the greatest outcomes in the industry.

In a 1993 study [8] modified a method known as simulated annealing to address a difficulty with shape-fitting. They studied two different formulations of the issue: one in which the shapes are reduced to grid-like structures, and another more exact formulation in which the shapes are collections of points on a plane. Additionally, they contrasted their new algorithm with another one they had created. Even if shapes overlap, they are initially arranged arbitrarily in the older technique. The chosen shape is then shifted around till it doesn't overlap any other shapes. They

discovered that, despite taking a little longer to complete, simulated annealing outperformed their previous strategy

Leao and his team examined various mathematical methods for fitting forms together in 2019 by [14], taking into account things like the components' orientation and the guidelines for making sure they don't overlap. They primarily concentrated on linear programming with mixed integers and nonlinear programming, two categories of mathematical models. How they handle rotating the pieces is the main distinction between the two. The model is linear if the rotations are preset or not taken into account. The math becomes more challenging and the model's behavior becomes nonlinear if the components may spin freely
A novel algorithm was created by [15] employing a branch-and-bound technique to expedite the arrangement of shapes (represented as polygons). They utilized a technique known as NFP to ensure that the shapes didn't overlap and sorted the shapes by their area in descending order. The CPLEX solver and other approaches were found to be up to 50% slower than their method.

To handle this NP-hard problem, various algorithmic approaches have been investigated, including bitmap representation with bitwise overlap checks and polygonal representation with no-fit polygons for overlap detection. Mixed-Integer Programming (MIP) models have also been developed, and their solution times have been improved using techniques such as lifting and branching strategies. These algorithmic solutions not only promise increased efficiency, but also speed availability, and lower training requirements as compared to human expertise [16].

Chapter 2

Proposed Heuristic Approaches

Heuristic methods to address the cutting and packing problem often follow a constructive approach, where items are iteratively inserted into the track and provide a feasible solution to the problem. However, there are still improvement procedures, which aimed improving the initial solution out of local search algorithms, decrease wasted space within the layout of the positioned piece by employing these enhancement procedures. For the problem of nesting with free rotation approached in In this work, we need to answer these three questions, when building a solution:

1st: In which series to sequence the parts?
2nd: what is the angle on which each part have to rotate?
3rd: How to position each part in the track?

To answer these questions, two position methods, five sorting methods and two rotation methods were implemented that is prpopsed in the literature. Each method was implemented without dependence on other methods. In the next segments, we present which representation of parts and range was used; which non-overlapping checking algorithm has been implemented. Furthermore, we will present the sorting rules used, rotation rules and placement rules. Than Finally, we implement the evolutionary algorithms to test the problem which is in particular, are Genetic Algorithm and Particle Swarm Optimization algorithm (PSO).

2.1 Representation of parts and strip

Each of the pieces is represented by an array of ordered points (x, y), where the point (x, y) is connected with the point (X), the last point is connected with the first. With this type of representation it is possible to define any polygon, just storing where each vertex is located, regardless of its complexity and number of irregularities. The representation used is exemplified in (Figure 2.1).

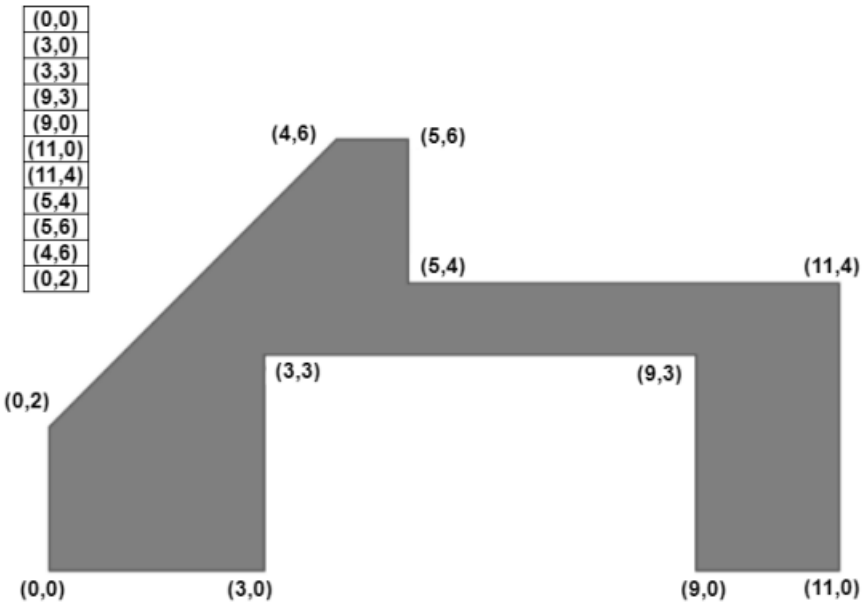

Figure 2.1 Representation of pieces by point vector

The particular representation scheme offers a space-efficient approach to store polygons, as it requires sorting only the vertices.

The range is defined by a constant "L"representing its width, along with a variable "W" representing the utilized height of the strip, which is sought to be minimized. Evaluationg the objective function value for a solution (i.e., the value of "W") involves examining all polygons to find one of the highest-valued vertices in the "y" coordinate.
Whenever a part is positioned, we verify that all its points fall withing the boundaries of the strip. For simplicity, we assume that the band is positioned in the first quadrant of the Cartesian plane (i.e.its lower left corner is positioned at (0,0)).

2.2 Checking For Overlapping parts

As discussed in section 1.4, there are several approaches to verify the overlapping of non-convex parts. Among these are the NFP [2], the No-fit Raster [9], straight lines and separating parabolas [4]. In this work, we opted to employ a method based on triangles. This decision was influenced by the challenges associated with using NFP and No-Fit Raster when dealing with the problem that involve free rotations. Given this point, we choose to use the triangulation of the pieces.

In the proposed method, each part is partitioned into triangles, in a pre-processing step. To check if two pieces overlap, we calculate the overlap between the triangles that make them up. Initially,

a rectangle is drawn around each component. When rectangles are created, it is checked to see if the parts collide using simple tests with the vertices positions. The parts are not overlapping if the rectangles do not overlap (Figure 2.2). But we can more efficiently use the shapely library and test if the area of their intersection is strictly positive (if it is zero, we could have the situation where they have a point or an edge in common, which is allowed).

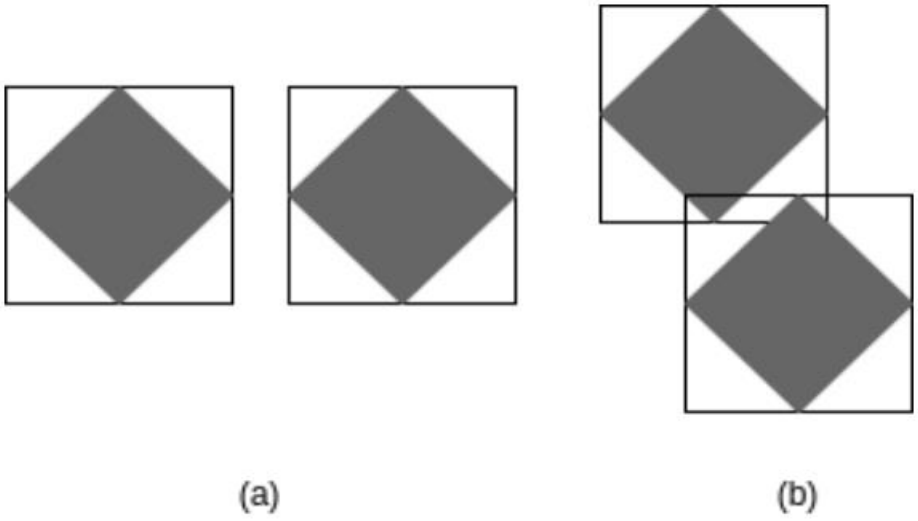

Figure 2.2 Test with the Enclosing Triangles

As can be seen in figure (2.2) the fact that the surrounding rectangles of the pieces are overlapping does not necessarily mean that the pieces are too. In cases where the rectangles overlap, we take a cautious approach by partitioning the parts into triangles and conducting further tests. Here we have to note that this verification using rectangles is not mandatory, but it significantly improves efficiency, leading to a reduced time spent on the overlap check.
An algorithm called earfold [17] is used to divide a polygon into a collection of triangles. Its purpose is to create the list of triangles that constitute the polygon. The idea is as follows: as long as the polygon has more than three vertices, we look for a vertex that is both a reflex and an ear, then we add the corresponding triangle to the list of triangles. At the end, three vertices remain which form a last triangle which is also added to the list.

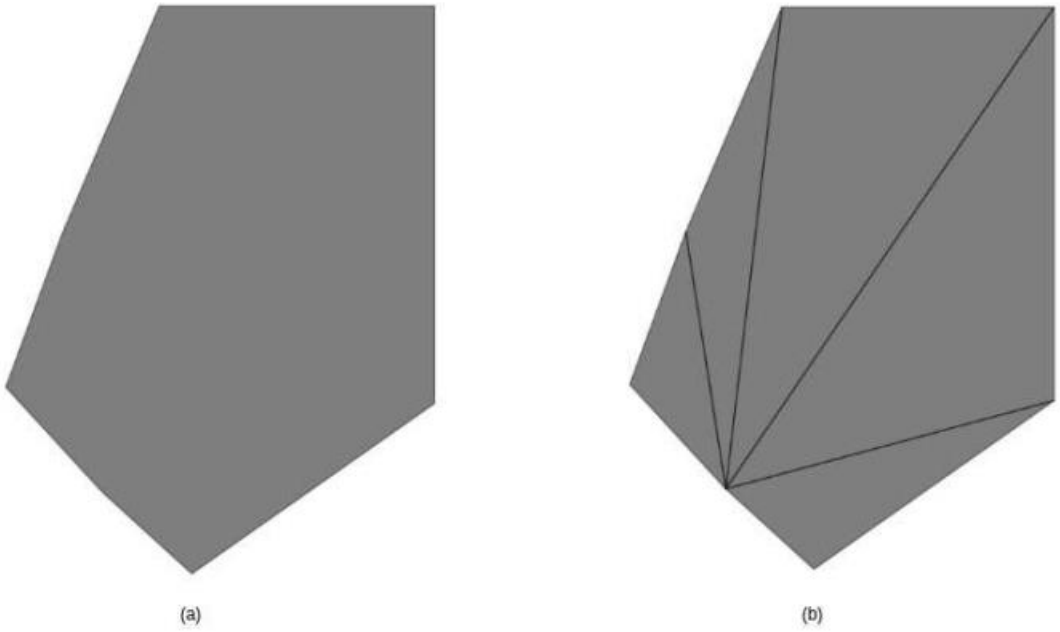

Figure 2.3 Polygon before and after triangulation

2.3 Sorting rules

The arrangement of the pieces in the strip can significantly alter the function's quality after the algorithm has been run. While the simple approach involves randomly sequencing the polygons. In this work, we adopt predefined ordering for polygons.

Four of the eight suggested (OA, OB, OD, OE) [7] pieces sequences rules were used in the work [18], in addition to a criterion (OC) introduced in [7]. The five criteria used in descending order are shown below.

For function A, we return the area.
For function B, we return the discrepancy between the polygons area and the bounded rectangles area.
For function C, we return the polygon area subtracted the enclosed rectangle area, all divided by the enclosed rectangles area.
For function D, we return the diameter.
For function E, we return the area of the enclosing rectangle.

or rather, in each case, the opposite of this quantity so that the sorting is done in descending order.

To illustrate the ordering criteria used, we observe Figure (2.4). In the third representation of Figure (3.4), OD is expressed in terms of the separation between the upper and the lower end points points of the enclosing rectangle, which are represented by A = (x_{min}, y_{min}) and B = (X_{max}, Y_{max}), respectively. The Cartesian plane Euclidean distance between the two points is used in this.

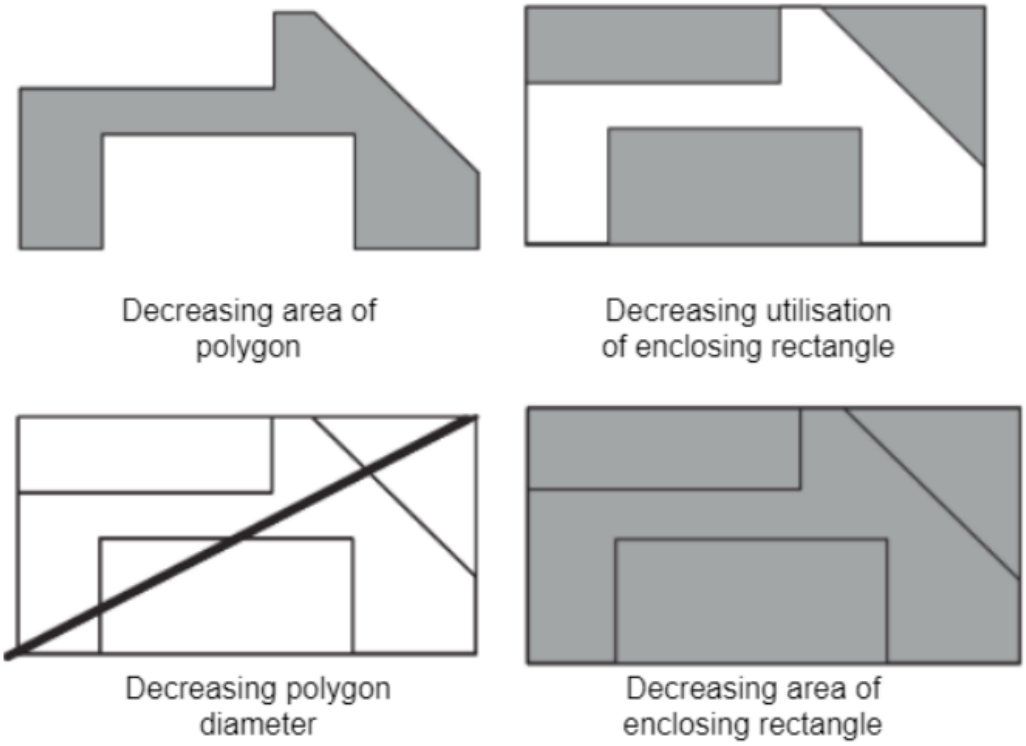

Figure 2.4 Ordering rules

The sorting rule proposed in the work [7], (OC) performs polygons sorting based decreasing usage percentage of the surrounding rectangle, while [18] use absolute values.

2.4 Rotation rule

The problem of nesting studied in the work [7], consider that the polygon can rotate at any angle θp, where e $0 \leq \theta p \leq 2\pi$. In the work [7] two rotation rules were proposed, which will be used for all polygons
The first one is signify by RA, taking into account the length and height of the pieces. The second rotation rule is denoted by RB, takes into account the descend of an border of the polygon.

2.4.1 Rotation rule Rα

The first rule is to rotate a quarter turn in the case where its length is greater than its height (where its two dimensions are calculated as usual with the coordinates of the extremal points of the polygon). After the polygons rotation is complete, we observe that now it will take up less space on axis , so the idea of this type of rotation is to make more polygons fit next to the rotated items. Not needed to place other polygons above it.

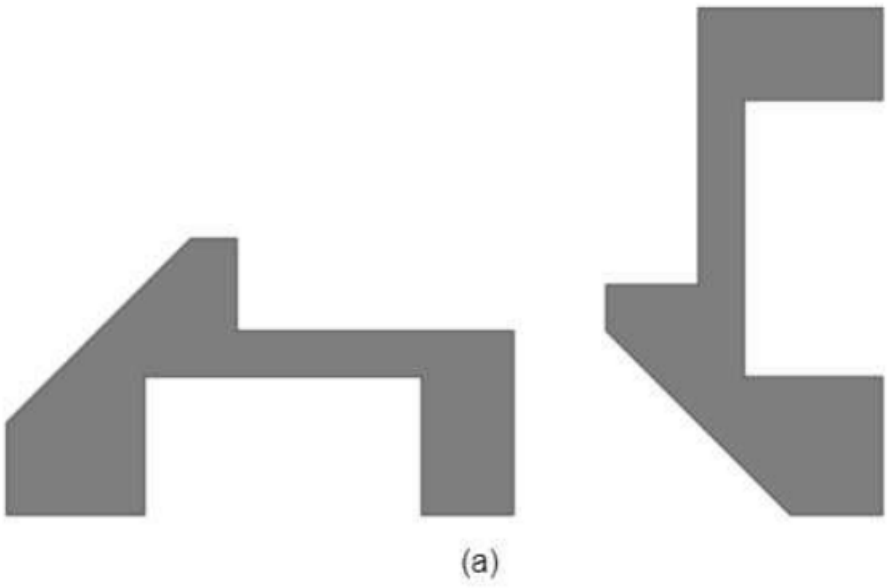

Figure 2.5 1st Rotation rule α

2.4.2 Rotation rule Rβ

It generalizes the above method and consider the descend of border of polygon. we select an edge and spin the polygon. The edge is chosen on some conditions and tests. the rule requires choice of an edge and the axis to be implemented. For this all the edges and the axis are tested. And the edge at which the evaluation function has the lowest value is chosen for the next step. The edge can be calculated using parameter M & N. where M is the absolute difference between the y-coordinates of the end point of the edge and N is the absolute difference between the x-coordinates of the end point of the edge.
The calculation of the angle depends on the value of M & N and it is given by different formulas based on conditions.

$$\Theta = \begin{cases} \arctan(N/M), & \text{if } N \geq M \\ \pi/2 - \arctan(M/N), & \text{if } N < M \\ \pi/2, & \text{if } M = 0 \\ 0 & \text{if } N = 0 \end{cases}$$

if N > = M the angle is tangent of N divided by M. However, if N < M the angle is difference between pi/2 and the tangent of M is divided by N. in case M = 0 the angle is equal to pi/2 if N = 0 the angle is equal to 0.
(Figure 2.6) represents a sample of a polygon derived from this rotation rule. In this particular case, the choosen edge of the polygon, which was previously tilted is now aligned in parallel to the y-axis.

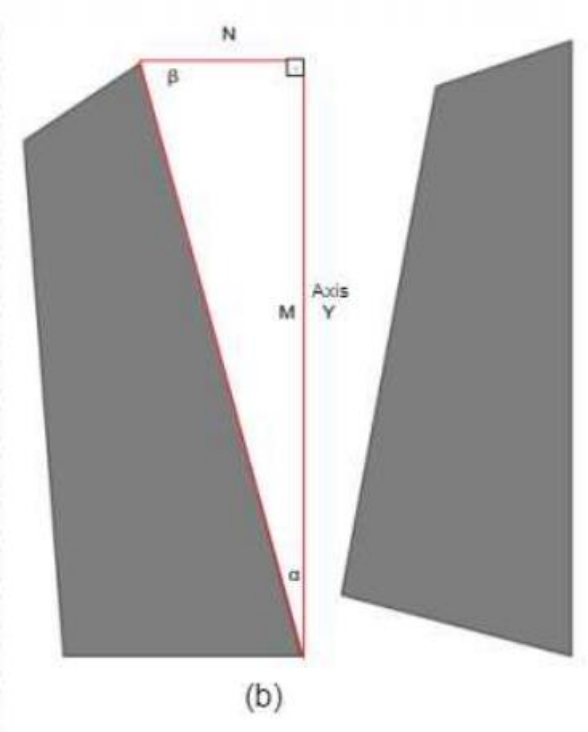

Figure 2.6 2nd Rotation rule β

2.5 Positioning rules

The way a part is inserted into the plate affects the use of unoccupied spaces, which can could be occupied by remaining pieces. In our work, the positioning methods are used to decide how a piece will place. When examinig the classical heuristics in the nesting literature, the work [7] proposed two rules for the positioning of the pieces.

2.5.1 Positioning rule Pα

The rule α considers all the possible combinations in which each of the vertices of the piece to be inserted are exactly at each of the vertices of the pieces already allocated. If there is a valid position, those in which the part to be inserted is lower is chosen. If there is more than one position, the one in which the piece is most near to the left is selected. After all the possible combination have been made, the invalid positions (which cause overlap with any already allocated piece or leave the piece out of range) will be eliminate.

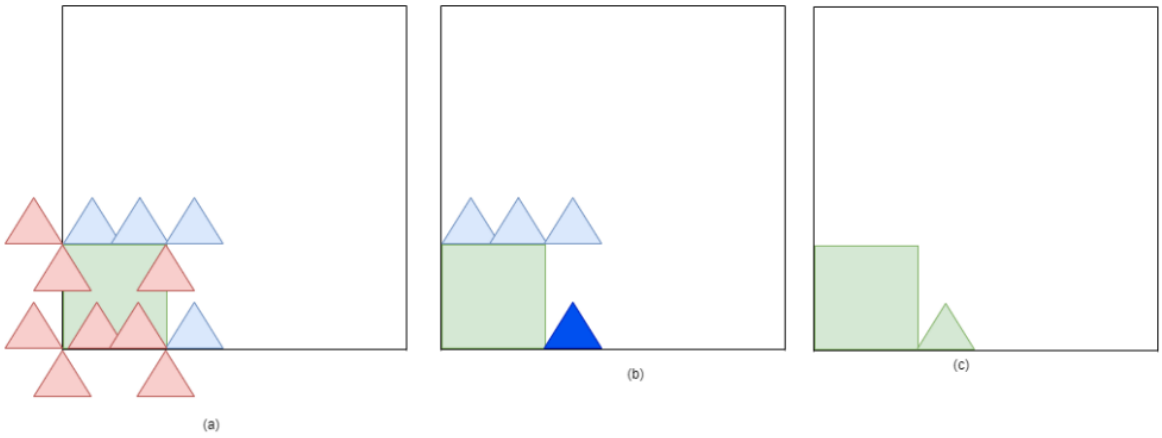

Figure 2.7 Position rule α

Figure (2.7, a) represents the vertices of the new piece are combined to the pieces that are already present in the solution. Figure (2.7, b) shows the invalid places are eliminate, leaving only the possible places of valid pieces. Next in Figure (2.7, c) the most suitable position for the new piece is selected, in which the part is lower and more to the left. The new piece is than inserted to the new position and the entire methods ends at this point. The process ensures an efficient placement of the new piece.

2.5.2 Positioning rule Pβ

The rule composed of starting a part at the y-axis and make an effort to locate it above the other components. After the position has been depict the piece is slide right and the procedure occur again. After all the pieces are entered. The invalid (which cause overlapping with any already allocated piece or leave the piece out of range) are eliminated. The places that makes the part be in the further down and leftmost is chosen, thus the piece is slide downward and to be left until it cannot slide to this position.

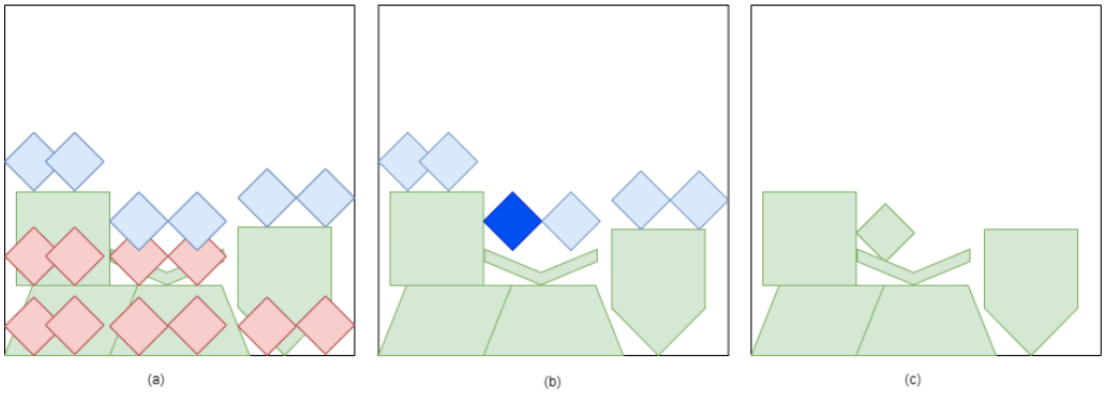

Figure 2.8 Position rule β

As can be seen in Figure (2.8, a), the part starts at the origin and is positioned on top of the parts that are above it. This sequence is repeated when the component is laterally moved to the right. As shown in Figure 2.8(b), when the component reaches the predefined range's border, the procedure comes to an end. All invalid placements are now removed. In figure (2.8, c), the place at which the part is most below and further to the left is chosen. Finally the piece is swiped downward and to the left and than it is placed there.

2.6 Genetic Algorithm

A meta-heuristic framework called the Genetic Algorithm (GA) is influenced by natural evolution. For encoding purposes, it makes use of the idea of individuals, represented as chromosomes. The algorithm includes features like elitism to choose strong individuals, crossover to share genetic information, mutations to introduce variety, and fitness assessment to adapt to the environment.

The GA was originally introduced by the [19] and after that many works adopt this approach. In the work [9] A Biased Random Key Genetic Algorithm is presented, for the two-dimensional nesting issue with fixed rotation constraints. Based on that work, this approach will be implemented in our study .
Search optimization in the context of genetic algorithms is carried out using two different approaches. The selection and crossing processes are part of the first, referred to as reproduction. In this case, parents from the existing population produce new offspring, conserving and combining the genotype and/or phenotype of the parental units. The second search process makes small adjustments to the current solution space to locate locations nearby that have higher objective function values.

These algorithms are designed to systematically replace substandard individuals with better ones to achieve convergence toward an ideal fitness score [20]. Both stochastically and with the aid of construction heuristics, the starting population can be instantiated. Reputable academics have reported considerable results when using random initialization, including [21], [22] and [23]. The effectiveness of heuristic-based initialization, on the other hand, has been shown in research by [24] and [25], which result in highly optimized solutions.

Below we explain a Pseudo code of a basic genetic algorithm by [20].

```
Begin GA
    Generate initial population
    WHILE NOT stop DO
      BEGIN
        Select parents from population Produce children from selected parents
        Mutate the children
        Extend the population adding children to it
        Reduce the extended population
      END
   Output the best individual found
END
```

Figure 2.9, Genetic Algorithm Pseudo Code

A fundamental Genetic Algorithm for optimization is described in the pseudocode. An initial population of potential solutions (referred to as "individuals") is created first. Once a stopping condition is met, a loop continues to run. The ability of "parents" to create "children" through crossover is taken into account while choosing them for each iteration. Then, to provide genetic variation, these kids are modified. The population is expanded by the children, and then it is whittled down to its original size by choosing the fittest individuals. The cycle repeats until a stopping requirement, such as a predetermined number of iterations or a predetermined level of fitness, is satisfied. The algorithm then returns the best possible result that it could have found while running.

2.7 Particle-Swarm-Optimization Algorithm

The collective behavior seen in natural systems, such as bird flocks and fish schools, served as the conceptual inspiration for the Particle Swarm Optimization (PSO) algorithm, a heuristic optimization technique. It is crucial to look at its underlying model, the Boid (Bird-oid) model, which was first put forth in 1987 [26], in order to understand its developmental trajectory and historical background. This computer framework, which attempted to mimic avian flocking behavior, was a crucial forerunner to the PSO algorithm. Each bird is represented as a point in the Boid model, with random initial velocity and location parameters, in a Cartesian coordinate system. The "nearest proximity velocity matching" idea underlies the simulation, ensuring that each player modifies their velocity to match their closest neighbor. The simulation quickly converges as it continues to iterate, resulting in uniform velocity at all places. This model, however, is noticeably oversimplified and lacking in empirical realism. A stochastic variable is added to the velocity term at each iteration to help alleviate this. This modification adds a degree of randomness, making the simulation more in line with actual events.

According to [27], each particle represents a potential solution and is identified by its position and velocity, which are typically initialized at random. To assess the quality of each particle's present position, the algorithm uses a fitness function. Every particle maintains its position as its personal best (pBest), and the best of these is known as the global best (gBest). Based on these pBest and gBest values, the swarm experiences iterative adjustments that move it closer to the ideal outcome. The algorithm repeats until a predetermined stopping criterion, such as a set limit on iterations or a set minimum error threshold, is satisfied [28]. PSO's versatility and effectiveness have been effectively demonstrated in a number of domains, including machine learning, robotics, and finance.

Below we explain a Pseudo code of a basic Particle-Swarm-Optimization algorithm by [29].

```
For each particle
  Initialize particle
END
  Do
        For each particle
          Calculate fitness value
 If the fitness value is better than the best fitness value (pBest) in history
      set current value as the new pBest
End
      Choose the particle with the best fitness value of all the particles as the gBest
       For each particle
       Calculate particle velocity according equation          (2)
       Update particle position according equation             (1)
  End
       While maximum iterations or minimum error criteria is not attained
```

Figure 2.10, Particle-Swarm-Optimization Algorithm Pseudo code

The Particle Swarm Optimization (PSO) algorithm, is described in the pseudocode by the procedural stages involved. The approach begins with the formation of a swarm of particles, each of which represents a potential solution in the multidimensional search space relevant to the optimization problem under consideration. For these particles, stochastic methods are often used to produce their initial locations and velocities. The PSO algorithm's main goal is to iteratively modify these particles' placements so that they move toward an optimal or nearly optimal solution. The algorithm uses fitness evaluations to measure the adequacy of each particle's current position in order to do this. The best position of each particle is recorded as its personal best (pBest pBest), and the best position among all of them is referred to as the global best (gBest gBest). In subsequent rounds, each particle's velocity and position are computationally updated using established equations that are often designed to take into account the particle's current velocity, pBest and gBest. Until a predetermined stopping criterion, such as a maximum iteration count or a minimum error threshold, is met, the algorithm continues in this repetitive process.

2.8 Test Instances

All the problems contain convex and non-convex pieces, and each can have a unit quantity or replicas. The source repository [30], a database of classic instances taken from literature, each Instance in dataset includes a fixed number of pieces and a strip length. In this context, the predetermined angles of some pieces are ignored due to the free rotation feature. Table 1 shows the instances names (column Inst), the number of pieces that each instance has (column P) and the dimension of the strip (column W). In our work with did not deal with the last three files due to their larger size.

TABLE I
INSTANCES USED.

Inst.	$\|P\|$	$L \times W$
albano	24	4900
blaz	24	15
dighe2	10	100
han	23	58
jakobs	25	40
jakobs2	25	70
mao	20	2550
marques	24	104
poly1a	15	40
shapes	43	40
shirts	99	40
trousers	66	79

Figure 2.11 Test Instances

Chapter 3

DEAP User Manual Tutorial

An open-source Python package called DEAP, or Distributed Evolutionary Algorithm in Python, is made for carrying out various evolutionary computation tasks. Strongly and weakly typed genetic algorithms (GAs), as well as multi-objective evolutionary algorithms (MOEAs) like NSGA-II and SPEA2, are among the many evolutionary algorithms that it supports. With the help of DEAP's key evolutionary computation features, users can create single- and multi-objective evolutionary algorithms and take advantage of the processing capacity of numerous CPU cores
DEAP is a framework for swift prototyping and ideas validation. In contrast to many existing frameworks, it takes a unique approach by making algorithms explicit and data structure clarity. It provides appropriate tools to users to make sure they easily write their own evolutionary loops. The latest release version of DEAP is 1.2.2 and supports both version of Python (2 and 3).
Additionally, it incorporates simple parallelism, which frees users from having to worry about implementation nitpicks like synchronization and load balancing [31].
Basic knowledge of Python is required in order to use DEAP library. DEAP contains working examples for beginners. DEAP does not contain Graphical User Interface (GUI), hence, it does not give facility of graphical plots [32]. The DEAP only supports one plot which is of individual geneology. The programming language contains limitations in terms of performance. Python is slower than compiled languages, such as C. DEAP is not best for computation of expensive fitness. DEAP can be run on Linux, Microsoft Windows, and Mac OS X. It needs Python 2.6 or above. It was introduced at GECCO 2012 [33].

3.1 Overview

We will notice that DEAP adopts a distinct strategy if we have used previous evolutionary algorithm frameworks. DEAP allows us the option to design and configure our own initializers and types rather than forcing us to use established ones. It does not force to use any certain operators; instead, it promotes deliberate decision-making. With DEAP, we can create algorithms that are customized to our unique requirements, unlike other frameworks that offer pre-defined algorithms.

In this research, GA and PSO (evolutionary algorithms) are implemented using DEAP.

3.2 Installation

DEAP supports Python 2.7 and 3.4 or higher. The computation distribution needs Scalable Concurrent Operations in Python (SCOOP). CMA-ES needs a Numpy library, and matplotlib library is recommended for the visualization of results as it is completely compatible with DEAP's API.

DEAP can be installed on the system using easy_install or pip. The easy_install deap or pip install deap are the commands to install DEAP on the system. Furthermore, python setup.py install can be used to build from sources, download or clone the repository, and type.

3.3 Types

Type is an important part of coding. User needs to use available types, DEAP gives the facility to make our own types. Creating a relevant type can usually be time-consuming but this can be done with the help of the creator which makes it easy. This can be accomplished in one line. For example, the given below code creates a FitnessMin class for a minimize function and an individual class that is created from a list with a fitness feature set to the just created fitness. (Figure 3.1) shows an example of type creation in DEAP. The degree of customisation DEAP gives is one of its distinctive benefits.

```
from deap import base, creator
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)
```

Figure 3.1 Example of type creation

3.3.1 Individuals

When we consider the wide range of evolutionary algorithms that are currently available, such as Genetic Algorithms (GA), Genetic Programming (GP), Evolutionary Strategies (ES), Particle Swarm Optimization (PSO), Differential Evolution (DE), and others, the possibilities for individual types are enormous. This is another evidence that developers cannot provide all possible types. The 'creator' module and 'Toolbox' of DEAP are helpful in this situation. They

allow us to customize the framework to meet our unique demands because they let us construct and initialize our own unique individual kinds.

3.3.2 Particle

A particle in the world of evolutionary algorithms is a special kind of person. Usually, it possesses qualities like speed and a memory of its ideal location. In DEAP, creating this individual is comparable to inheriting from a list. We add attributes for maximum and minimum speed as well as ideal position. Additionally, we can register an initialization function, such as "initParticle()," which accepts the particle class, size, domain, and speed limitations as arguments. A fully-formed particle with a speed vector and a fitness attribute intended to maximize two goals is returned when we call "toolbox.particle()". This demonstrates how DEAP may adapt to different personality types.

3.3.3 Swarm

A swarm is a distinct entity in Particle Swarm Optimization (PSO) that has a communication network among its particles. A completely connected network is the easiest to understand because every particle in the swarm is aware of the highest position it has ever attained. This is typically done by copying the global best position to a 'gbest' attribute and the global best fitness to a 'gbestfit' attribute. We obtain a fully-formed swarm when we execute "toolbox.swarm()." The algorithm should update the 'gbest' and 'gbestfit' characteristics to reflect the most recent best position and fitness after each evaluation. This demonstrates DEAP's adaptability to various optimization techniques.

3.4 Initialization

Setting up initial values is the next stage implementation after establishing data types. This procedure is made simple in DEAP. The Toolbox serves as a storage area for several tools, including initializers with specific functions. We can simply generate initializers for certain individuals near the conclusion of the program, which are essentially collections of random floating-point values.

With the help of individuals that have already been preloaded with random float values, these initializers can provide functions that initialize populations. These features are given particular names and added to the toolbox along with their default settings. For instance, we can quickly produce a complete population by using the function "toolbox.population()". For a demonstration of how initialization functions in DEAP, see (Figure 3.2).

```
import random
from deap import tools

IND_SIZE = 10

toolbox = base.Toolbox()
toolbox.register("attribute", random.random)
toolbox.register("individual", tools.initRepeat, creator.Individual,
                 toolbox.attribute, n=IND_SIZE)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

Figure 3.2 Initialization in DEAP

3.5 Evolutionary Tools

An extensive set of operators necessary for the operation of evolutionary algorithms can be found in the Tools module. The modification, choice, and movement of the individuals within their computational environment depend heavily on these operators. This module, which is part of the Toolbox framework, adds features like Hall-of-Fame, Statistics, and History to the core methods in addition to fundamental operators.

3.6 Using the Toolbox

The toolbox serves as a comprehensive collection of evolutionary tools encompassing everything from object initializers to evolution operators. Its primary purpose is to facilitate the seamless configuration of each algorithm. To achieve flexibility and extensibility, the toolbox primarily relies on two main methods: register () and unregister (). These methods play a vital role in adding and removing tools from the toolbox, enabling easy customization and adaptation to various evolutionary processes. Functions like mate (), mutate (), evaluate (), and select () are frequently included in the standard nomenclature for operators in evolutionary tools. The system does, however, allow for the registration of any name that may be uniquely associated with these operators. The Toolbox acts as the repository for these operators, which are called using their generic identities for building evolutionary algorithms.

3.7 Operators

The operators are used to transform or select individuals. For instance, when two individuals are supplied to the crossover operator, those two entities are transformed in place. Operators carries out crucial operations that either change or choose people. For instance, the crossover operator directly modifies two individuals if we pass them both through. However, it is the user's responsibility to make sure that any progeny are distinct from their parents and lose their fitness. The majority of the time, algorithms accomplish this by duplicating a person using the toolbox.clone() function and then invalidating the fitness values by using the 'del' command. Some already implemented operators are available in the tools modules of DEAP. Once we select our required operator we have to register it in the toolbox. The list of the implemented operators available in DEAP is shown in (Figure 3.3).

Initialization	Crossover	Mutation	Selection	Migration
initRepeat()	cxOnePoint()	mutGaussian()	selTournament()	migRing()
initIterate()	cxTwoPoint()	mutShuffleIndexes()	selRoulette()	
initCycle()	cxUniform()	mutFlipBit()	selNSGA2()	
	cxPartialyMatched()	mutPolynomialBounded()	selNSGA3()	
	cxUniformPartialyMatched()	mutUniformInt()	selSPEA2()	
	cxOrdered()	mutESLogNormal()	selRandom()	
	cxBlend()		selBest()	
	cxESBlend()		selWorst()	
	cxESTwoPoint()		selTournamentDCD()	
	cxSimulatedBinary()		selDoubleTournament()	
	cxSimulatedBinaryBounded()		selStochasticUniversalSampling()	
	cxMessyOnePoint()		selLexicase()	
			selEpsilonLexicase()	
			selAutomaticEpsilonLexicase()	

Figure 3.3 Deap Operators

3.8 Multiple Processors

SCOOP is a distributed task module that allows parallel programming on multiple environments, ranging from heterogeneous grids to supercomputers. Its interface is similar to the concurrent. futures module which was introduced in Python 3.2. It has two main functions submit() and map() which distribute computation efficiently and easily over a grid of computers. Working on the multiprocessing module is similar to working on SCOOP. It is done by replacing the relevant function with the distributed one in the toolbox.ant function with the distributed one in the toolbox.

Users can configure and execute GP in an intuitive manner with the help of DEAP. It implements its EA using pluggable genetic and selection operators. Users create different flavours of GP and GA by combining different parts at API level. Multiple genetic operators are implemented in DEAP. (Figure 3.4) shows code snippet.

```
from deap import creator
from deap import gp
...

creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", gp.PrimitiveTree, fitness=creator.
    FitnessMax)

toolbox = base.Toolbox()
toolbox.register("expr", gp.genHalfAndHalf, pset=pset, min_=1,
    max_=2)
toolbox.register("individual", tools.initIterate, creator.
    Individual, toolbox.expr)
toolbox.register("population", tools.initRepeat, list, toolbox.
    individual)
toolbox.register("compile", gp.compile, pset=pset)
toolbox.register("select", tools.selTournament, tournsize=3)
toolbox.register("mate", gp.cxOnePoint)
toolbox.register("expr_mut", gp.genFull, min_=0, max_=2)
toolbox.register("mutate", gp.mutUniform, expr=toolbox.expr_mut,
    pset=pset)

toolbox.register("evaluate", myEval, args1=arg_list1, arg2=
    arg_list2)

toolbox.decorate("mate", gp.staticLimit(key=operator.attrgetter("
    height"), max_value=10))
toolbox.decorate("mutate", gp.staticLimit(key=operator.attrgetter(
    "height"), max_value=10))
...
```

Figure 3.4 An example of evolutionary Loop in Deap

3.9 Major feature

Deap is a python framework that can be executed on linux, microsoft windows and Mac OS X platforms. It is compatible with Python 2.6 or any latest version.

By default the library supports array-like representations and tree-based representations enabling users to perform genetic programming (GP). Deap offers an intuitive and declarative approach to configuring and executing genetic programming.
At its core, DEAP implements an evolutionary algorithm that allow users to use pluggable genetic and selection operators. By combining different components at the API level, user can create various type of genetic programming and genetic algorithm [34].
DEAP includes a wide range of commonly used genetic operators that are available for the users to use.

3.10 Statistics

Users can record statistical actions using the Statistics object, a flexible tool that works similarly to a toolbox. Its unique combination of Python's flexibility and capability with a simplified core of transparent Evolutionary Computation (EC) components makes it stand out. By making the majority of the pieces explicit, this design fosters creativity while making it simpler to swiftly prototype new Evolutionary Algorithms (EAs). That's not all, though. The framework also gives capabilities for simple parallel computing with just a few code changes. This implies that we can divide the labor-intensive computational tasks among several machines in a cluster. Because there is no hierarchy and each worker node can create new tasks and share the overall effort, the distribution model is rather democratic. Even with complicated algorithms, the framework has shown to be user-friendly. Just two years after its debut, DEAP is already being used by researchers in variety of domains, from exploring bloat control in genetic programming to employing genetic algorithms to optimize sensor network location. Additionally, it is open source and freely accessible at http://deap.googlecode.com., making it available to anyone who wants to dive in [31].

3.11 Weaknesses

DEAP has a lot of power and flexibility, it also has significant disadvantages. [34] For the applications, which require costly fitness computation Deap maybe not a good choice. Fortunately, the creators have offered examples to aid newcomers. Second, compared to systems like jMetal [32].
DEAP lacks a Graphical User Interface (GUI) and offers a limited amount of graphical charting functionality. It only allows for the mapping of individual ancestry. The performance of Python itself can also be a bottleneck. It's often slower than compiled languages like C because it's an interpreted language, which could be a big problem for computationally demanding workloads.

3.12 Strengths

Regardless of our level of skill, DEAP is highly recommended for both Genetic Programming (GP) and other evolutionary algorithms, despite its drawbacks. DEAP provides fundamental data structures, genetic operations, and simple examples for GP beginners to help them running the code quickly. The versatility of DEAP enables for more sophisticated capabilities, such as fitness evaluation using parallel graphics hardware, if we are already knowledgeable about GP. It's a flexible tool that can be used by both novices and specialists [34].

Chapter 4

Genetic Algorithm Implementation

4.1 Development Environment

In this chapter, we lay down the algorithms we implemented in order to attain the more optimized results for our test problems. DEAP (Distributed Evolutionary Algorithms in Python) is the main Module we use to for the implementation of the algorithms. For the implementation of the algorithms Python language is used.

4.2 Genetic Algorithm

In this work, we use DEAP framework to implement our evolutionary algorithms. Genetic algorithms are fascinating to study and use because of the intricacy that results from these "simple" rules. Minor adjustments might produce surprising and intriguing results [35].
There are several algorithms implemented in the algorithms module of deap. The algorithms use a toolbox. In order to setup a toolbox for an algorithm, we must have to register the desired operators under the specified names. We work with "algorithms.eaSimple", The simple evolutionary algorithm takes 5 arguments as algorithms.eaSimple(pop, toolbox, cxpb, mutpb, ngen). These are the arguments which we use:

algorithms.eaSimple(population, toolbox, cxpb=0.5, mutpb=MUTATION_PROBABILITY, ngen=NUMBER_OF_GENERATIONS, stats=stats, halloffame=halloffame, verbose=True).

We start by defining four parameters for the genetic algorithm, which can be changed at will, and which have an impact on both the speed of execution and the quality of the solution found.

- the number of generations (the higher the better but the slower),
- the size of the population (the higher the better but the slower),

- the elitism rate (i.e., the proportion of the population that survives from one generation to the next),
- the mutation probability (that a given individual will be changed/mutated).

Which corresponds to the following code available [36] in DEAP Documentation:

```
creator.create("Fitness", base. Fitness, weights= (-1.0,0))
#weights--1.0,0 because we want to minimize the fitness and ignore the second value (rotation rule) creator.create("Individual", list, fitness-creator.Fitness)
# Create the toolbox
toolbox base. Toolbox()
# Attribute generator (random positioning rule)
toolbox.register("attr_positioning rule", random. choice, POSITIONING RULES) #Individual generator (List of positioning rules)
toolbox.register("individual", tools. initRepeat, creator. Individual, toolbox.attr_positioning_rule, len(pieces)) # Population generator (List of individuals)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
#Evaluation function toolbox.register("evaluate", evaluate, pieces-pieces, strip_width=strip_width) #Mate function (crossover)
toolbox.register("mate", tools.cxOnePoint)
#Mutate function toolbox.register("mutate", tools.mutShuffleIndexes, indpb-MUTATION_PROBABILITY) #Select function
toolbox.register("select", tools.selTournament, tournsize=TOURNAMENT SIZE) #Scoop
toolbox.register("map", futures.map)
```

4.3 Fitness Function

A fitness function is essential for assessing how well a given solution represented by a chromosome performs in solving a particular problem in optimization and evolutionary algorithms [37]. To evaluate the fitness of an individual, i.e., a series of positioning rules among alpha and beta, the pieces are positioned by applying them, first with the rotation rule A, then with the rotation rule B and the one of the two that gives the smallest strip height is retained. The fitness value is then this minimum strip height. The difficult task of positioning the parts is done in the position_pieces function.

We also need to return which rotation rule is the best one in order to later display the solution with matplotlib or simply to determine the coordinates of the pieces.

```
if heightA < heightB:
  return heightA, "A"
        else:
  return heightB, "B"
```

4.4 Positioning rules and the position_pieces function

The part position function is one of the most important (and complex) in this program. It takes as parameters the list of parts (i.e., polygons), the position rules to be applied (rules) and the width of the strip on which the parts will be arranged.

I start by making a copy of the parts, as they will be moved during the execution of the function. Then the chosen rotation rule is applied to all pieces (and in the case of rule B to all edges of all the pieces along each of the two axes X and Y).

The first piece is placed as the bottom left corner of the strip. Then, for each of the pieces (say of index i), we will find its position using the rule given by rules[i] (which is either alpha or beta):

- Initialize the list of all possible positions to [np.array([0, y_max])] so that it is not empty where y_max is the maximum of the y coordinates of all the pieces already placed: it is always possible, even if it is not the best solution, to place a polygon above all the others,
- Generate the positions according to the rule; for alpha we consider all the vertices of all the previous pieces, for beta we consider all the positions that are above the previous pieces, as well as the same positions but translated to the right,
- Filter to keep only the positions such that the polygon does not intersect the previous polygons and such that it does not overflow the strip,
- Choose the position which leads to the lowest strip height used.

At the end of the run, a position for all the pieces has been found and they can be returned.

4.5 Load pieces from a file

All the files in the test_instances folder have the same simple structure. The load_pieces function is a parser that extracts the value of strip_width from the first line of the file and the coordinates of the vertices for each polygon along with the quantity. The values returned at the end of the execution are the polygons (i.e., instances of the Polygon class) and the width of the strip.

We start by loading the pieces for a given file as explained above and by sorting them according to one of the sorting functions we have defined. Then, we give the description of our problem to the deap module. When building evolutionary algorithms, we used the toolbox i to contain the operators, which are later on called using their generic names.

4.6 Generation of individuals

A single indiviual is a single solution. Individuals combine two types of solutions, The chromosome, which contains the basic 'genetic' information (genotype) that the GA is looking for to deal with it, and the phenotype which is the somatic presentation of chromosome [38].
The individuals in the population are going to be generated: by calling **random.choice(POSITIONING_RULES)** for each piece, that is len(pieces) times. The properties of an individual is a list of alpha and beta, (string of characters) one for each piece, one of them is **"ALPHA"**, the other one is **"BETA"**. If the first is **"ALPHA",** than alpha is applied to the 1st piece , if the fifth is **"BETA"** then beta is applied to the fifth piece .

Example**. [random.choice(POSITIONING_RULES)** for _ in range(len(pieces))] is similar to the generation done by deap and returns individuals such as ['BETA', 'ALPHA', 'ALPHA', 'BETA', 'ALPHA'] or ['ALPHA', 'BETA', 'ALPHA', 'BETA', 'BETA'].

4.7 Population

A population is a group of persons. A population is made up of the individuals being tested, the phenotypic attributes that define the individuals, and some information about the search area. The population size for each problem is determined by the problem's complexity. A random population initialization is frequently used [38].
Similarly in DEAP, the population is created by repeatedly creating individuals:

population = toolbox.population(n=POPULATION_SIZE)

Here population is the variable that will hold the **population** of an individual. **Toolbox.population** is a function withing the toolbox that is resposible for generating the population of an individual. Than n=POPULATION_SIZE is an argument passed to population function. It specifies the number of individuals to be created in the population.

Finally, we link the functions we have defined so far (such as evaluate) to the module or we use built-in function.

4.8 Evaluation

The evalution process decodes a chromosome and assign it a fitness measure [39]. The eval(V) funtion assigns a score to each alternative solution, or chromosome based on how finest it is. The better a solution is, the more likely it is to be chosen for the following round [38].
The evaluation is the most personal part of an evolutionary algorithm, it is the only part of the library that we must have to write by ourself. A typical evaluation function takes one individual as argument and returns its fitness as a tuple. A fitness is called a list of floating point values and

has a reasonable characteristic to know if this individual shall be re-evaluated. The fitness is set by setting the values to the associated tuple.

The evaluate function i have defined takes the individual whose fitness we are computing but also two extra arguments: the list of pieces to place on the strip and the width of the strip. Both are loaded from the test file as described above.

4.9 Crossover stage

In Genetic algorithm crossover step also called recombination, is a process of combining the genetic information of two parents to generate new generation [40].

4.9.1 Two Point Crossover

Specifically, for the crossover step, we use the tools.cxTwoPoint. In the deap documentation one can read:

deap.tools.cxTwoPoint(ind1, ind2)

Executes a two-point crossover on the input sequence individuals. The two individuals are modified in place and both keep their original length.

Given two individuals, we split their genome into three parts (that is we make two cuts), say respectively [A, B, C] and [A', B', C']. Then, their crossover is either [A, B', C] or [A', B, C'] randomly.

In a two-point crossover, two crossover locations on the parent chromosomes are chosen at random in place of one. The offspring is then produced by swapping the bits between these two places between the parents. Two-point crossover basically entails performing two single-point crossovers at various locations along the chromosome. With the addition of 'k' crossover points, this idea can be expanded to allow for even greater genetic diversity. The goal is to combine and match more chromosomal segments from the parents, possibly creating kids with better characteristic combinations [40].

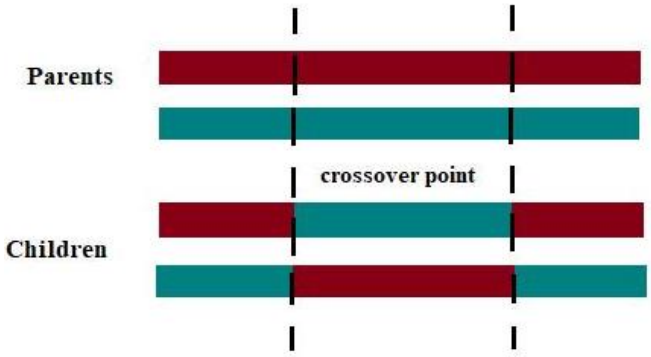

Figure 4.1 Representaion of Two point crossover

But for more experiments we also run it with some other crossover operators of deap, such as cxOnePoint() crossover and cxUniform() crossover.

4.9.2 One Point Crossover

To execute a one-point crossover without any further changes in the code. It would split the parents into respectively [A, B] and [A', B']. Then, their crossover is either [A, B'] or [A', B] randomly.

The 'crossover point' is chosen at random on both parents' chromosomes. [40],The two parents then switch off the parts to the right of this point. Due to this, two children are produced, with each receiving genetic material from both parents. In order to produce children that are more adapted to their environment, the procedure seeks to combine the best characteristics from each parent.

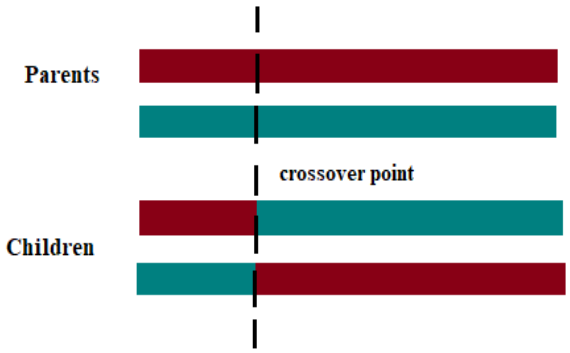

Figure 4.2 Representation of one point crossover

4.9.3 Uniform Crossover

The uniform crossover operator is extremely usefull in genetic algorithms specifically when the likelihood of gene inclusioni s fine-tuned [41].

Each gene is taken into account individually rather than in segments in uniform crossover. A coin is essentially flipped for each gene to determine which parent it will come from. This indicates that there is an equal chance that every piece in the offspring might have come from either parent. But various mixing ratios can also be applied. This raises the possibility that one parent may have provided the child with more genetic material than the other. This mechanisim offers a great deal of flexibility and randomness, which might produce children with a variety of traits [40].

Finally halloffame is a sorted list used to store the best individuals found during the execution of the algorithm, that is the sets of positioning rules that lead to the smallest strip heights.

4.10 Mutation Stage

Genetic Algorithm has always a low mutation rate because high mutation rates convert (GA) to a primitive random search. The mutation operator maintain population variety by adding another level of randomization. In reality, this operator prevents similar solutions from emerging [42].
To meet the requirements of diverse evolutionary algorithm implementations, the Distributed Evolutionary Algorithms in Python (DEAP) module offers a variety of mutation operators. The MutFlipBit, MutShuffleIndexes, and MutUniformInt mutation operators are among them. We go into great depth about each of these mutation operators below.

4.10.1 MutShuffleIndexes Mutation

For the Mutation step we use the **tools.mutShuffleIndexes**.

In the deap documentation one can read:

deap.tools.mutShuffleIndexes(individual, indpb) Shuffle the attributes of the input individual and return the mutant. The individual is expected to be sequence. The indpb argument is the probability of each attribute to be moved. It is especially helpful when changing people who are represented as a list of integers or other ordinal values. The operator shuffles the individual's indexes according to a specific mutation probability **indpb.**

Parameters:

Individual: we want to alter this sequence in some way. This sequence's components and attributes can all be switched around.
The probability that each element will be relocated to a different place in the sequence is represented by the integer indpb, which ranges from 0 to 1.

What it Returns: The function produces a new sequence (a "mutant") in which some of the elements may have been rearranged or shifted. This new sequence has a tuple around it [43].

For further experiments we use use the **mutFlipBit** and **mutUniformInt** mutation operators .

4.10.2 MutFlipBit Mutation

The Distributed MutFlipBit operator is made for mutating objects that are represented as binary strings. It swaps the input individual's attribute values and returns the mutant [44] Based on a user-defined mutation probability indpb, each bit in the individual's genotype is "flipped" (changed from 0 to 1 or from 1 to 0).
This function accepts two inputs: a person and a probability value. The function examines each personal characteristic and determines whether to flip (alter) it depending on the supplied probability ('indpb').
It provides a single-item tuple containing the modified person [45].
The random decisions for flipping the attributes are made by using the 'random()' built-in function in Python.

4.10.3 MutUniformInt Mutation

When dealing with individuals who are represented as sequences of numbers, the MutUniformInt operator is intended to be used. With a user-defined mutation probability (indpb), this operator replaces each integer value in the individual with a random integer taken at random from a uniform distribution and falling within a predetermined range. [High, low]. By replacing attributes with an integer that is uniformly distributed between low and up inclusively, MutUniformInt modifies an individual with probability indpb [46].
Here, the range from which random integers will be generated is specified by low and high, and indpb is the independent likelihood that each attribute will change.

Parameters:

Individual: This is the thing we need to change. It is a list of attributes, such as an integer list.

low: This indicates the lowest value that any attribute could possibly have following a mutation. A single value that applies to all attributes can be provided, as can a list of values, one for each attribute.

Up: Like Low, this indicates the maximum value that any attribute can have followed a mutation.

The likelihood that each specific attribute will change is known as the indpb. It has a value ranging from 0 to 1.
Returns:
A single-element tuple containing the altered individual is returned by the function.

4.11 Selection Stage

DEAP provides a broad range of built-in selection operators, which are essential for evolutionary algorithms since they control how individuals are selected for the following generation [47].

4.11.1 SelBest Selection

For the selection step we use the tools.selBest from the deap documentation **Deap.tools.selBest()** Selects the k best individuals among the input individual. The list returned contains reference to the input individual. Based on their fitness value, the selBest operator chooses the top individuals from the population. In other words, it chooses the top k people, sorted by fitness, where k is the total number of people we want to choose [48].

For our further experiments with the Genetic algorithm we also use the **selTournament()** and selRandom() selection operator of DEAP.

4.11.2 SelTournament selection

The selTournament operator chooses a few people at random from the entire population to compete in a tournament. Three primary parameters govern the operator:
individuals: a population at large from which to choose.
k: The number of candidates to be chosen.
tournsize: The total number of competitors in each tournament.
For every one of the k choices:
Pick individuals of random sizes from the entire population. Depending on their level of fitness, choose the contestant who is the best. The winner should be included in the group of chosen people [49]. Using larger tournaments increases selection pressure since below average individuals are less likely to win a tournament, but above average persons are more likely to win [37].

4.11.3 SelRandom Selection

A selection method called the selRandom operator chooses people at random from a specified population. SelRandom is entirely stochastic, in contrast to other selection techniques like selBest or selTournament that include an individual's fitness value while making a choice. When we need to add unpredictability to the selection process, it is most frequently utilized

The following parameters are typically passed to the selRandom operator:

Individuals: A list of potential candidates from which to choose.
k: The number of candidates to be chosen.

The operator selects k people at random from the list without taking their fitness values into account [50].

4.12 Main Loop

The eaSimple function from the deap module is used to run the genetic algorithm. Its arguments are:

- the initial population (a list of individuals),
- the toolbox that defines all the operators,
- the mating probability (one half),
- the mutation probability,
- the number of generations,
- the statistics to be tracked,
- the hall of fame to be filled.

In my case, we use the following stats argument:

stats = tools.Statistics(lambda individual: evaluate(individual, pieces, strip_width)[0])

stats.register(“avg”, np.mean)

stats.register(“max”, np.max)

stats.register(“min”, np.min)

to track the average, best and worst individuals for each generation.

We can then get the best overall individual by looking at the first element in the hall of fame: best_individual = halloffame[0].

Chapter 5

Particle Swarm Optimization Algorithm Implementation

5.1 Particle-Swarm-Optimization

Particle Swarm Optimization (PSO) has experience many modifications since it is introduce in 1995 [51]. Particle Swarm Optimization (PSO) is a technique inspired by the social behavior of birds and fish. In PSO, a group of potential solutions, called particles, navigates through a multidimensional search space in order to find the optimal solution to a problem. Each particle continually refines its position, guided by the bestperforming particles in the swarm. This process iterates until a satisfactory solution is found.

In a vast area filled with possibilities, they are all scouting out the ideal location or solution. The fact that they collaborate with one another and share their discoveries is fantastic [52]. The finest speed of convergence that PSO's method exhibits for single-objective optimization makes it seem particularly well suited for multiobjective optimization [53].

In our context, the search space encompasses all conceivable positional rules applicable to the pieces. To ensure our solutions are represented as arrays of numeric values, we use the value 0 to signify the alpha rule and 1 to represent the beta rule, instead of using the string labels ALPHA or BETA.

The code responsible for printing the solution becomes the following:

```
print (list ("ALPHA" if r < .5 else "BETA" for r in best))
```

5.2 Population size

In Particle Swarm Optimization (PSO) depends on the problem but is typically not very sensitive to changes in the problem. Larger populations are utilized in particular instances and range in size from 20 to 50 for standard sizes. Setting up the first population is essential. Along with more sophisticated techniques, random generation is frequently used by [54]and Orthogonal Design [55]. These techniques strive for a uniform initial distribution, which improves the efficiency of search space exploration and optimization of solutions.

5.3 The update_particle function

The update_particle function is essential to the PSO algorithm. Using conventional PSO update equations, it recalculates each particle's velocity and location based on recent data and the world's best position. For effective search and solution optimization, this feature is essential [56].

The crucial part for this optimization technique is updating a particle’s position given:

- the best fitness value in the swarm and,
- $\phi 1$ the personal learning coefficient,
- $\phi 2$ the social learning coefficient.

It proceeds as follows:

- Generate two random vectors, u1 and u2 with as many coordinates as pieces to be placed.
- Calculate two vectors, vu1 and vu2, by element-wise multiplication of u1 and u2 with the difference between the best position and the current position of the particle.
- Update the particle’s speed by adding vu1 and vu2 to the current speed vector.
- Clip the elements of the speed vector to ensure they stay within a predefined range.
- Update the particle’s position by adding the updated speed vector to the current position.

which corresponds to the following Python 3 code see the DEAP documentation at [57].

```
def update_particle(part, best, phi1, phi2):
    u1 = [random.uniform(0, phi1) for _ in range(len(part))]
    u2 = [random.uniform(0, phi2) for _ in range(len(part))]

    v_u1 = list(map(operator.mul, u1, map(operator.sub, part.best, part)))
    v_u2 = list(map(operator.mul, u2, map(operator.sub, best, part)))

    part.speed = list(map(operator.add, part.speed, map(operator.add, v_u1,
v_u2)))

    for i, speed in enumerate(part.speed):
        # Clip the speed to keep it between smin and smax
        if abs(speed) < part.smin:
            part.speed[i] = math.copysign(part.smin, speed)
        elif abs(speed) > part.smax:
            part.speed[i] = math.copysign(part.smax, speed)

    part[:] = list(map(operator.add, part, part.speed))
```

Figure 5.1 Update Particles

5.4 The Main Loop

For each generation (that is GEN times), we have to perform the following steps:

- Iterate through each particle (part) in the population.
- Evaluate the fitness of the particle using the evaluate function.
- Check if the particle's current best fitness is less than its current fitness and update it if needed.
- Check if the global best fitness is less than the particle's current fitness and update it if needed.
- Update the position of all particles using the update_particle function described above.

In Python, this is done as follows:

```
for g in range(GEN):  # for each generation
    for part in pop:  # for each particle
        part.fitness.values = toolbox.evaluate(part)
        if not part.best or part.best.fitness < part.fitness:  # Update the value
of the best position if needed
            part.best = creator.Particle(part)
            part.best.fitness.values = part.fitness.values
        if not best or best.fitness < part.fitness:  # Update the value of the
overall best position if needed
            best = creator.Particle(part)
            best.fitness.values = part.fitness.values
    for part in pop:
        toolbox.update(part, best)  # Update the value of the position of all
particles
    logbook.record(gen=g, evals=len(pop), stats=stats.compile(pop))
    print(logbook.stream)
```

Figure 5.2 Main Loop for PSO Algorithm

5.5 Further experiments

The current version of the code uses:

toolbox.register("update", update Particle, phi1=1.0, phi=1.0)

But we perform many experiments with the (PSO) algorithm by increasing the personal learning coefficient ($\phi 1$) to emphasize an individual particle's exploration of its own solutions or by increasing the social learning coefficient ($\phi 2$) to make particles rely more on information from their neighbors. In addition, increasing the number of generations and/or individuals allows for a more extensive search process, potentially leading to better optimisation results. However, this will also increase the runtime of the code.

Chapter 6

Experiments

6.1 Datasets

Datasets contains shapes that could appear as a single object or in multiples and had both simple and sophisticated forms. These test databases were from [30], which has well-known literary samples. The database lists the number of pieces and strip width for each sample. The width of the plate is fixed for all the datasets. Below we describes all the datasets in details along their shape, that is used in our work.

6.1.1 Albano

The study done in 1980 by Albano and Sappupo focused on a textile-related issue. They concentrated on textile patterns with a variety of shapes, such as polygons and non-polygonal pieces with arcs. They approximated these designs using polygonal forms after scanning sample layouts from a paper source. The 24 participants in this study provided insightful contributions to the design and layout approaches used in the textile sector [30].

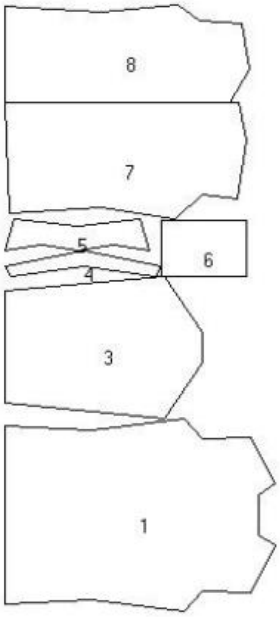

Figure 6.1 Dataset Albano Shapes

6.1.2 Blaz

In the BLAZ2 test set, the emphasis was on a layout with four different way of piece, a total of 20 pieces to be put in the proper order. Each component has roughly 7.5 vertices on average. The investigation permitted the placement of these parts in two practicable orientations: one at 0 degrees and the other at 180 degrees. The layout was done on a plate that was 15 units wide. This example gave a thorough look at how to optimize arrangements for a fair number of diverse parts [30].

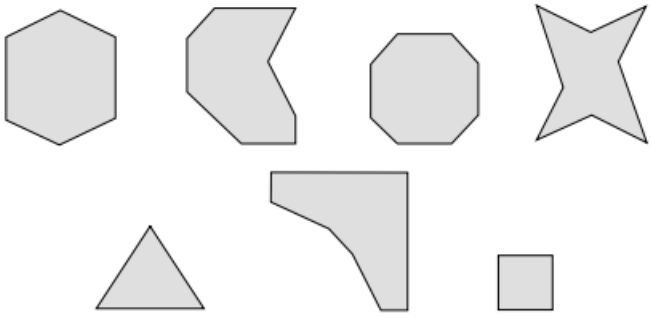

Figure 6.2 dataset Blaz Shapes

6.1.3 Dighe2

Jigsaw puzzles were the subject of a 1996 study by Dighe and Jakiela, known as "Dighe2." The example they looked at dealt only with polygons and had a layout size of 10. The data for this investigation, in contrast to earlier studies that employed scanned layouts, was created using a sample layout that was provided in the paper. The study offers an intriguing twist from the more typical textile optimization challenges by using computational approaches to the solution of jigsaw puzzle configurations [30].

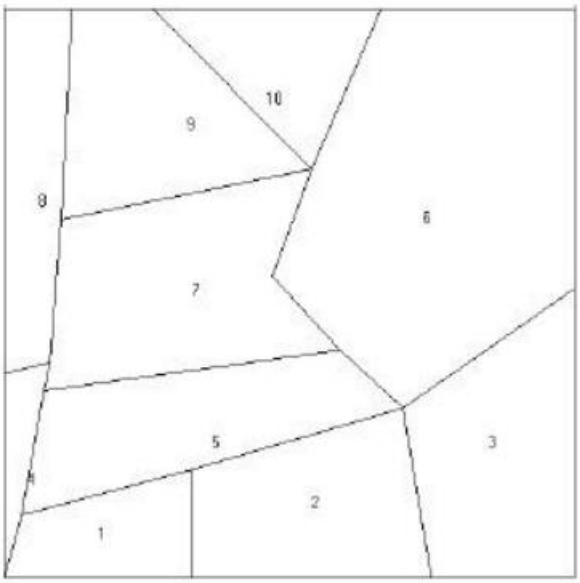

Figure 6.3 Dataset of Dighe2 Shapes

6.1.4 Jakobs2

In a 1996 paper by Jakobs, referred to as "Jackobs," the research focused on a fictitious issue requiring a layout size of 25. The data was created using a sample layout that was supplied in the article, and the shapes in question were all polygons. The inclusion of a "factor 2," albeit the precise use of this factor is not covered here, was a novel aspect of this study [30]. This research stands out for its concentration on manufactured challenges, providing an alternative perspective to the computational techniques typically used to address real-world problems.

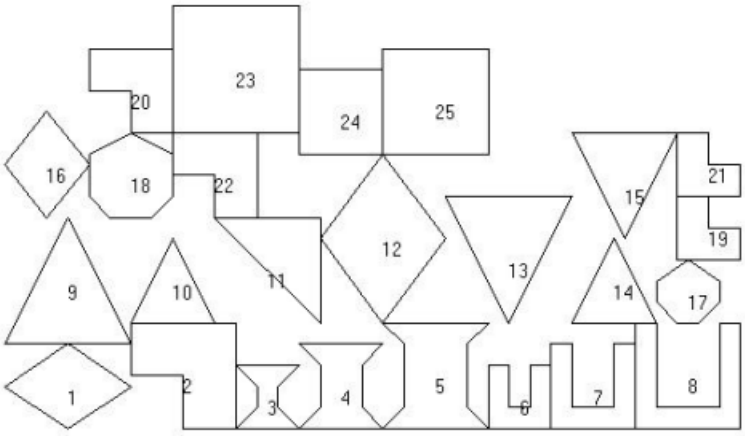

Figure 6.4 Dataset Jakobs2 Shapes

6.1.5 Mao

In a 1997 study, Bounsaythip and Maouche focused on cutting patterns for complex forms like polygons and non-polygonal pieces with arcs in order to address optimization issues in the textile sector [30]. For better analysis, they approximated these forms as polygons using data from scanned sample layouts. The size parameter for the study was set at 20 and included a "factor 5" element. This study proposes a computational solution to practical difficulties in the textile production industry.

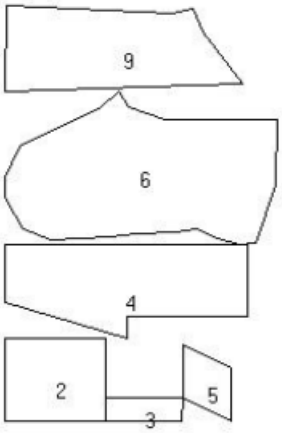

Figure 6.5 Dataset of Mao shapes

6.1.6 Marques

Similar to the work of Bounsaythip and Maouche, Marques et al. investigated optimization issues in the textile sector in a 1991 publication. The study, titled "Marques," focused on a layout size of 24 and sought to optimize the positioning of various shapes, including polygons and non-polygonal components with arcs. In order to make computation easier, data for the study was first estimated as polygons from scanned sample layouts. The research concentrated on using computational tools to tackle actual textile difficulties rather than mentioning a particular element [30].

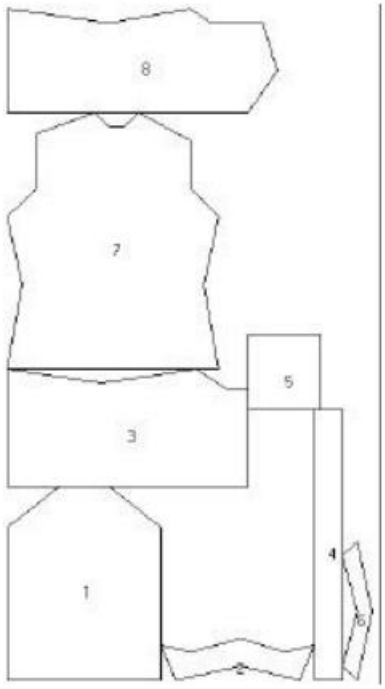

Figure 6.6 dataset Marques Shapes

6.2 Experiment Environment

We used Elektra server to conduct experiments in this report. Elektra machine is a dual processor machine in which each processor is Intel(R) Xeon(R) Silver 4214 CPU running at 2.2 GHz. Each processor has 12 cores where each core is 2-way hyper-threaded. In order to use the physical cores only, we can execute 12 threads per processor or 24 threads in total on both processors. But if we want to utilize hyper-threading as well, we can execute 24 threads on a single processor, or

48 threads in total on both processors. Elekrra machine has 93 GB of RAM and it is running Ubuntu 22.04.2 LTS operating system.

6.2.1 Genetic Algorithm Parameter settings

Here I use eaSimple algorithm from deap moduLe. For crossover I use the cxTwoPoint crossover operator, for mutation, I use the MutFlipBit operator and for selection, I use the selBest operator from the deap module.

The following possibilities were used:

- Number of generations: {20, 40, 60, 80, 100},
- Size of the population: {10, 15, 20, 25, 30},
- Mutation probability: {0.01%, 0.5%, 0.1%, 10%, 25%}.

6.2.2 PSO Algorithm Parameter Settings

We start the experiments with our Particle-warm-optimization (PSO) algorithm. In this phase, we use the following settings:

- Number of generations {20, 40, 60, 80, 100},
- Number of populations {10, 15, 20, 25, 30},
- Smin & smax values {(-1,1), (-2.2), (-3,3), (-5,5), (-3,3)},
- Phi1 & phi2 values {(1.0, 1.0), (2.0, 2.0), (3.0, 3.0), (4.0, 4.0)}.

Here phi1 is the personal learning coefficient and phi2 is the social learning coefficient.

6.3 Results of Genetic and Particle-Swarm-Optimization Algorithms

This section evaluates the performance of Genetic algorithm and Particle-swarm-Optimization Algorithm implemented for the Nesting problem with continuous rotation. I took 7 classic test problems from [30]. The section 6.3.1 includes the results tables and section 6.3.2 shows the graphical representation of the best solutions found in each execution. The tables in each experiments section presents the outcomes of utilizing the proposed genetic algorithm and PSO algorithm to solve various instances. The instance column displays the name of each instance. The Tot gen column indicates the number of generations established for the resolution of each particular instance, whereas the instance column displays the nomenclature of each individual case. The median result that the algorithm produced at the time the best solution was discovered is represented by the Med column. The Bt gen column indicates the generation at which the

genetic algorithm was successful in identifying this optimal solution, while the Bt sol column indicates the numerical magnitude of the most advantageous solution found. The Wt gen column corresponds to the generation in which this less-than-ideal solution was found, whereas the Wt sol column displays the numerical value of the least desirable solution found among all the created individuals.

6.3.1 Results

Table 1

Best and worst solutions of Genetic algorithm						
Instance	Tot gen	Med sol	Bt sol	Bt gen	Wt sol	Wt gen
Albano	20	14347.1	14347.09	0	27710.1	3
Albano	21	14482.6	14347.1	1	27710.1	21
Albano	60	14688.35	14347.08	2	27710.1	11
Albano	80	27657.4	14347.08	2	27710.1	6
Albano	100	27710.1	14347.08	0	27710.1	3
Best and worst solutions of PSO algorithm						
Instance	Tot gen	Med sol	Bt sol	Bt gen	Wt sol	Wt gen
Albano	20	16714.9	14347.09	0	27710.1	5
Albano	40	22349.6	14347.1	0	27294.1	28
Albano	60	26073.7	14347.08	0	27710.1	24
Albano	80	27657.4	14347.1	0	27710.1	8
Albano	100	27710.1	14347.1	0	27710.1	5

Table 2

Best and worst solutions of Genetic algorithm						
Instance	Tot gen	Med sol	Bt sol	Bt gen	Wt sol	Wt gen
blaz	20	38.4	33	0	120	5
blaz	40	67.53	33	1	120	26
blaz	40	113	33	3	120	26
blaz	80	120	32	0-4	120	5
blaz	100	120	32	0	120	5
Best and worst solutions of PSO algorithm						
Instance	Tot gen	Med sol	Bt sol	Bt gen	Wt sol	Wt gen
blaz	20	34.3	33	0	120	6
blaz	40	35.2	33	0	112	36
blaz	60	33.75	33	0	120	20
blaz	80	38.4	32	0	120	9
blaz	100	65.2	30	0	120	5

Table 3

Best and worst solutions of Genetic algorithm						
Instance	Tot gen	Med sol	Bt sol	Bt gen	Wt sol	Wt gen
dighe2	20	190.55	184	0	407.03	2
dighe2	40	189.15	184	3	407	20
dighe2	60	184	184	3	407.029	39
dighe2	80	184.8	184.0	0-7	407.03	3
dighe2	100	184	184.0	0	407.03	2
Best and worst solutions of PSO algorithm						
Instance	Tot gen	Med sol	Bt sol	Bt gen	Wt sol	Wt gen
dighe2	20	185.4	184	0	407.03	2
dighe2	40	189.15	184	0	407.03	23
dighe2	60	184.53	184	0	407.03	4
dighe2	80	184.8	184	0	407.03	3
dighe2	100	184	184	0	407.03	2

Table 4

Best and worst solutions of Genetic algorithm						
Instance	Tot gen	Med sol	Bt sol	Bt gen	Wt sol	Wt gen
Jakobs2	20	50.74	30	0	232	5
Jakobs2	40	191.46	30	3	232	16
Jakobs2	60	229.9	32	4	232	12
Jakobs2	80	232	33	0	232	6
Jakobs2	100	232	32	0	232	3
Best and worst solutions of PSO algorithm						
Instance	Tot gen	Med sol	Bt sol	Bt gen	Wt sol	Wt gen
Jakobs2	20	35	30	0	232	6
Jakobs2	40	35.86	32	0	232	40
Jakobs2	60	36.55	33	0	232	14
Jakobs2	80	36.92	32	1	232	8
Jakobs2	100	38.7	32	0	232	5

Table 5

Best and worst solutions of Genetic algorithm						
Instance	Tot gen	Med sol	Bt sol	Bt gen	Wt sol	Wt gen
mao	20	3017.73	2855.86	0	9580.78	3
mao	40	6909.66	2759.0	2	9580.01	39
mao	60	9428.75	2991.8	2	9580.78	57
mao	80	9580.78	2855.85	1	9580.78	6
mao	100	9580.78	2759.25	0	9580.78	3
Best and worst solutions of PSO algorithm						
Instance	Tot gen	Med sol	Bt sol	Bt gen	Wt sol	Wt gen
mao	20	3125.58	2900.41	0	9580.78	5
mao	40	3035.34	2854.88	7	9056.41	37
mao	60	3006.97	2855.85	2	9580.78	16
mao	80	3209.92	2900.40	0	9580.78	10
mao	100	3199.59	2820.03	0	9580.78	7

Table 6

Best and worst solutions of Genetic algorithm						
Instance	Tot gen	Med sol	Bt sol	Bt gen	Wt sol	Wt gen
marques	20	104.95	98	1	548.39	3
marques	40	402.82	103	1	548	37
marques	60	533.22	90.0	0	548.39	11
marques	80	548.38	102.0	0	548.39	2
marques	100	548.38	98.0	0	548.39	4
Best and worst solutions of PSO algorithm						
Instance	Tot gen	Med sol	Bt sol	Bt gen	Wt sol	Wt gen
marques	20	103.9	102	0	548.39	7
marques	40	108.46	98	0	516.39	32
marques	60	108.8	98	2	548.39	24
marques	80	104.28	98	0	548.39	7
marques	100	109.78	98	0	548.39	5

Table 7

Best and worst solutions of Genetic algorithm						
Instance	Tot gen	Med sol	Bt sol	Bt gen	Wt sol	Wt gen
Poly la	20	25.2	24	1	116.99	4
Poly la	40	24.85	24	0	116	8
poly la	60	23	26.0	0	116.99	8
Poly la	80	25.10	25	0	116.99	2
Poly la	100	25.25	26.0	0	116.99	2
Best and worst solutions of PSO algorithm						
Instance	Tot gen	Med sol	Bt sol	Bt gen	Wt sol	Wt gen
Poly la	20	30.5	24	0	116.99	5
Poly la	40	27.10	26	0	109.95	40
Poly la	60	28.92	26	0	116.99	15
Poly la	80	26.6	24	1	116.99	8
Poly1a	100	27.28	26	0	116.99	4

In this experiment phase our results show that increasing the number of generations generally leads to best performance, as more time allows the algorithm to explore more thoroughly. Such as for instance dighe2 instance find batter solution with 100 generations. For “blaz” instance it find the best solution at a very starting generation. However, increasing the population size had a more mixed effect, with some settings leading to improved performance while other did not. For the execution time in this section its not taking too long, as it is completing its all executions in almost 1 hour which is less time as compare to the other experiments which I made with other PSO algorithm settings. In larges instances such as “Jakobs2”, few generations were executed. As they are larger instances, with very complex parts. In “albano”instance When it reach to its 21st generation it stop its execution and gives the error “ lost of future track” and stop it execution after this we try many times to see if i twill complete it execution but the performance will be same it terminates after some generations.

In the experiment section with PSO “blaz” file takes long time than all other files. “Polyla” and “dighe2” complete its execution very fast with all the different settings I use. The other files execution is almost same, but I notice that when I increase the number of generations it takes too much time. With 100 number of generations each file takes 3 to 4 hours to complete its execution. And many times, they interrupt during their execution stating the error “connection reset” than I have to run them again. I also notice that Genetic algorithm goes well than PSO when we increase the number of generations. Even with 100 number of generation genetic algorithm did not take as much time as compared to PSO algorithm. I also try some experiments with higher value of smin and smax but it is not able to complete its execution with higher value specifically more than -5 and 5. In deap documentation the suitable value is -3 and 3. Here we also notice that

max is not very close to the min, so the variation quality between different solution is not very low.

The "Albano" exhibits a minimal negative deviation of approximately -0.0139%, while "Mao" and "Marques" display slightly larger negative deviations of approximately -0.952% and -0.784%, respectively. A negative deviation means that these algorithms tend to produce results that are slightly lower than the reference values. On the other hand, "Blaz" demonstrates a notable positive deviation of approximately 6.67%, indicating a significant overestimation. "Dighe2" and "Jakobs2" display very small positive deviations of approximately 0.00054% and 0.646%, respectively, while "Poly la" showcases a positive deviation of about 2.37%. A positive deviation means that these algorithms tend to produce results that are slightly higher than the reference values. The percentage deviation given by [7] is:

$$\text{Percentage Deviation} = \left|\frac{Xi-Yi}{Yi}\right| \text{ x } 100$$

In the specific context where X represents the output from a Genetic Algorithm (GA) and Y represents the output from a Particle Swarm Optimization (PSO) algorithm, we can interpret the sign of the deviation as an indicator of which algorithm is performing relatively better. A positive deviation for X would suggest that the Genetic Algorithm (GA) tends to produce results that are higher or better than the Particle Swarm Optimization (PSO) algorithm (Y), signifying that GA is performing relatively better. Conversely, a negative deviation for X would suggest that the GA tends to produce results that are lower or worse than the PSO algorithm (Y), indicating that PSO is performing relatively better. However, it's important to consider not only the sign but also the magnitude of the deviation, the specific goals of your optimization, and the application's requirements for accuracy and precision when assessing algorithm performance.

6.3.2 Graph Representation

The performance of Genetic Algorithm and Particle Swarm Optimization is stated in graph below. The vertical axis displays the quality of the solutions; the lower the value, the better. The horizontal axis represents the number of "generations". As a result, we can see how each method is performing, as they both attempt to make improvements to their solutions.

The graph shows the performance of Particle Swarm Optimization with Genetic Algorithm for five different number of generations, 20, 40, 60, 80, and 100. The green bars for Particle Swarm Optimization do the same as the blue bars for the performance of the Genetic Algorithm at each of those places. It demonstrates how the performance of the two approache after a number of iterations. It's a terrific technique to determine which is more useful for the issue we're attempting to address.

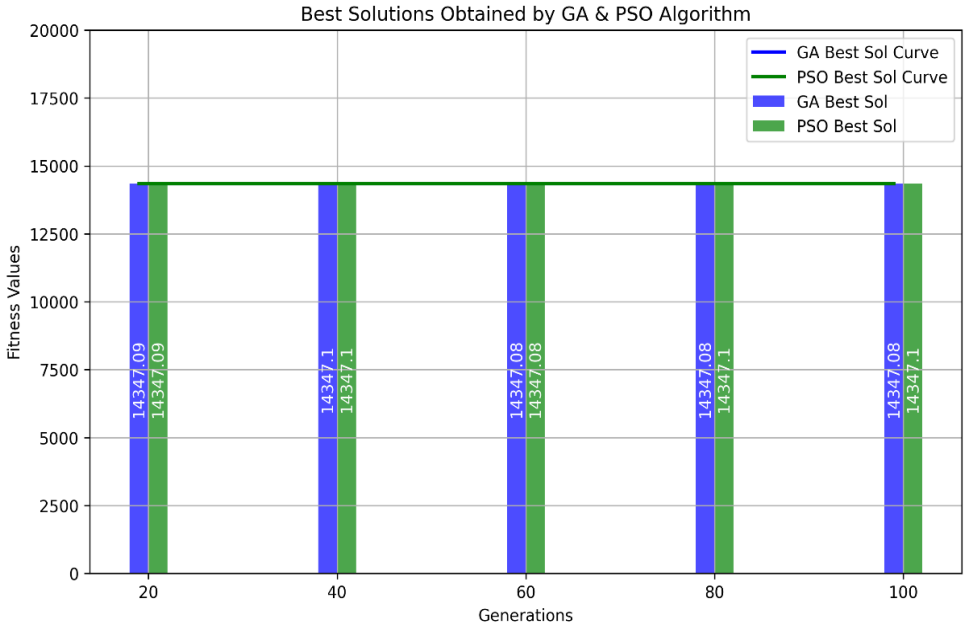

Figure 6.7 Albano Dataset best solutions

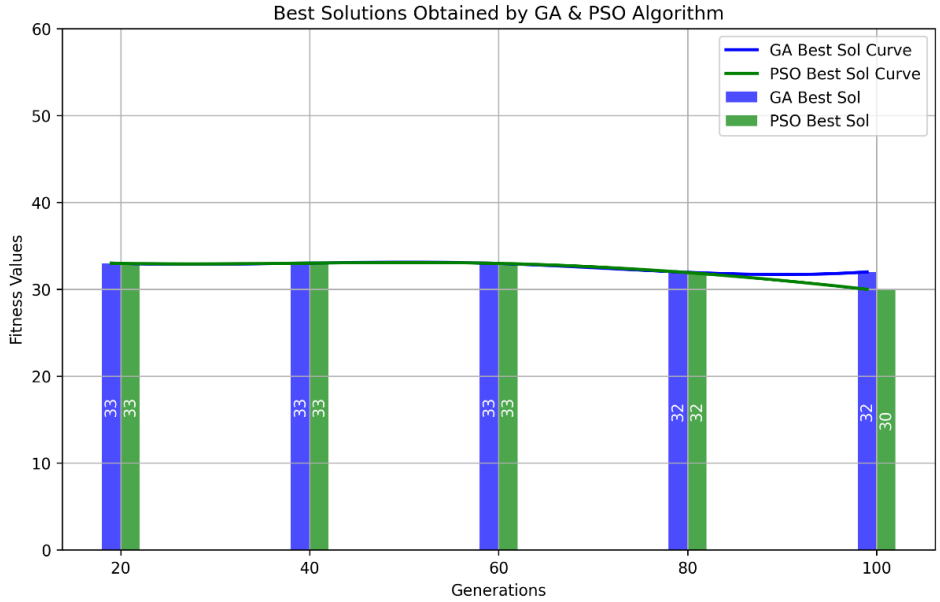

Figure 6.8 Blaz Dataset best solutions

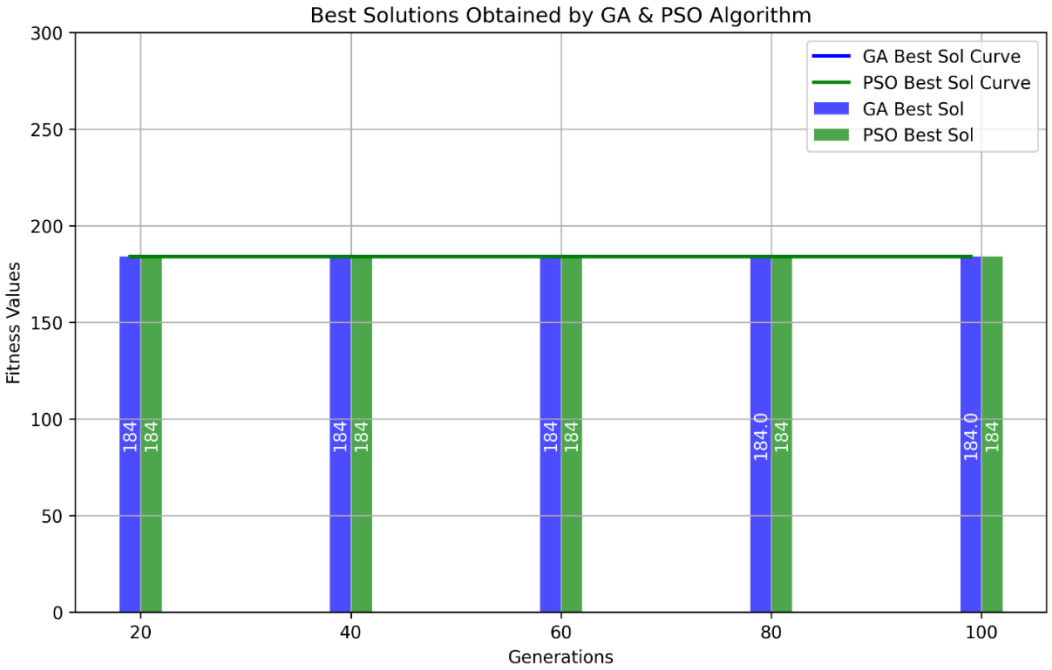

Figure 6.9 Dighe2 Dataset best solutions

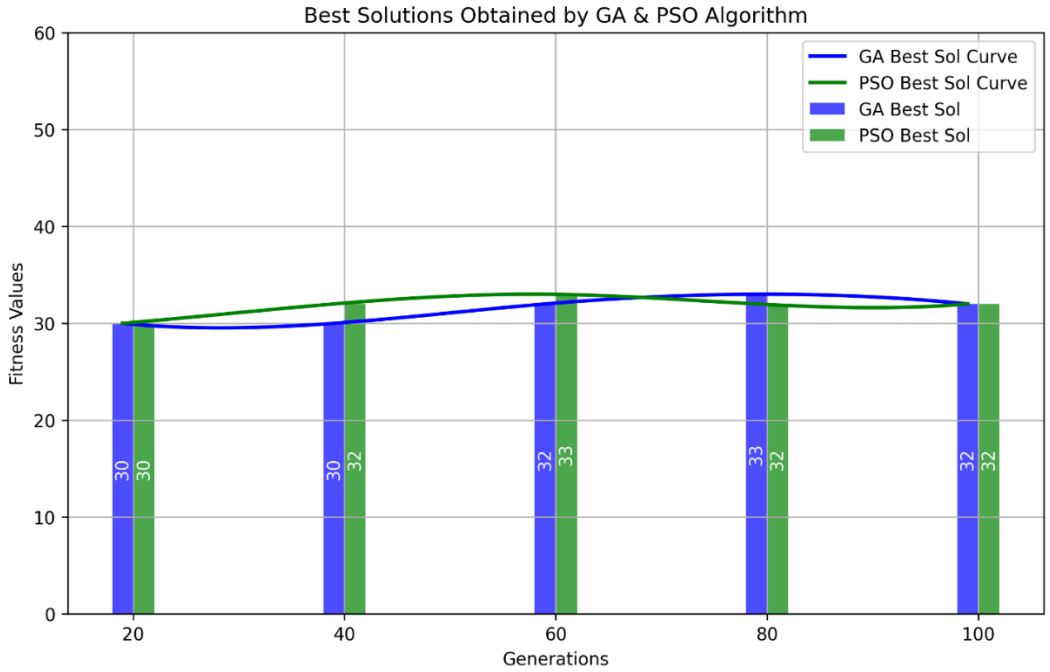

Figure 6.10 Jakobs2 Dataset best solutions

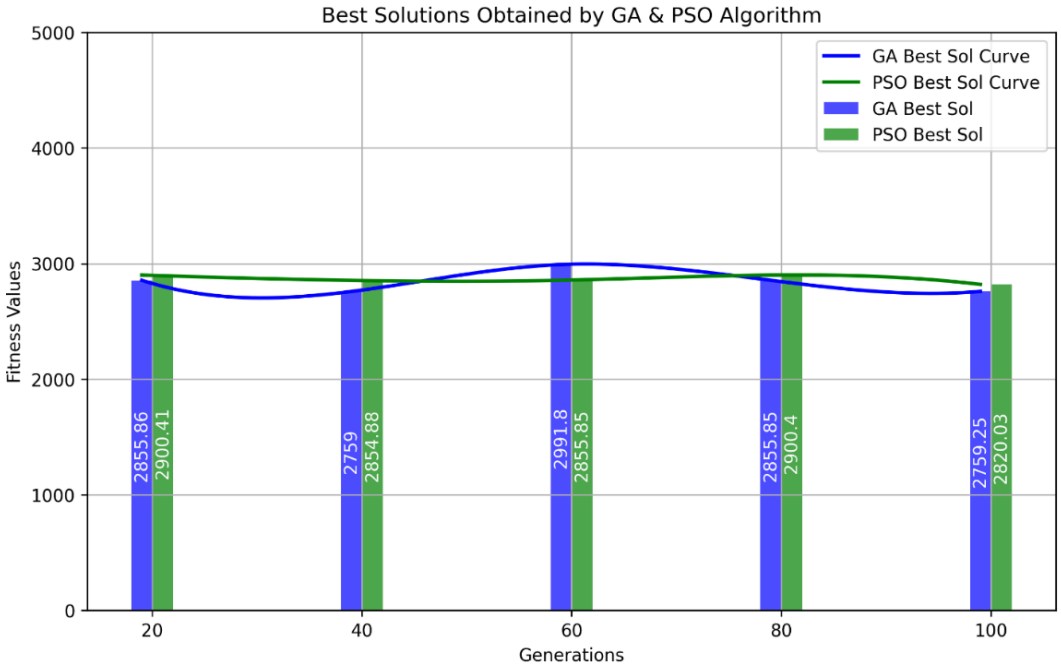

Figure 6.11 Mao Dataset best solutions

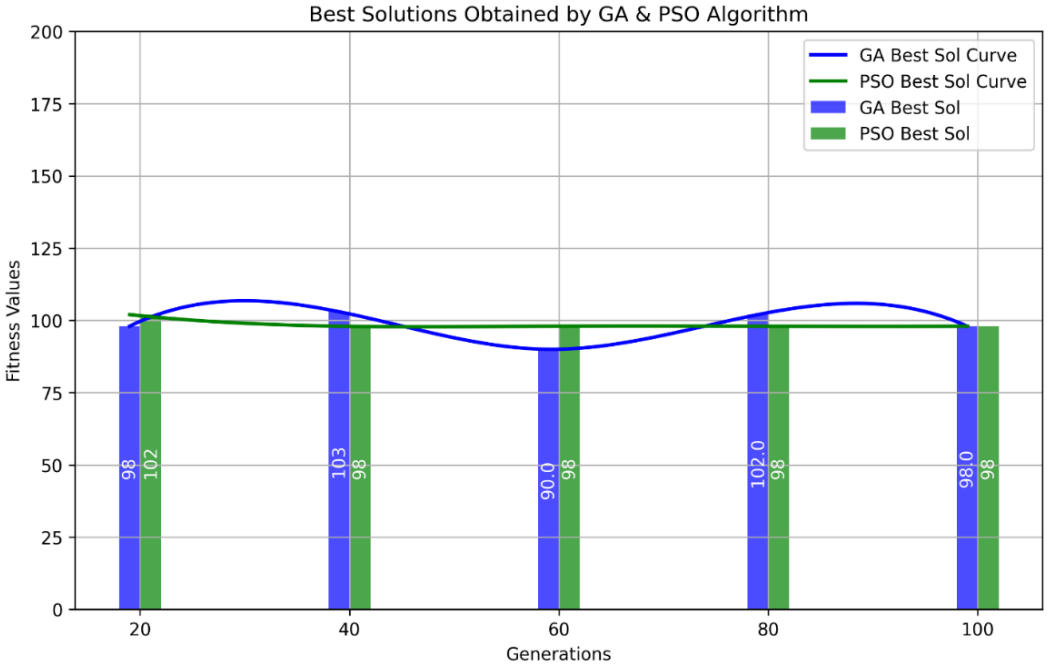

Figure 6.12 Marques Dataset best solutions

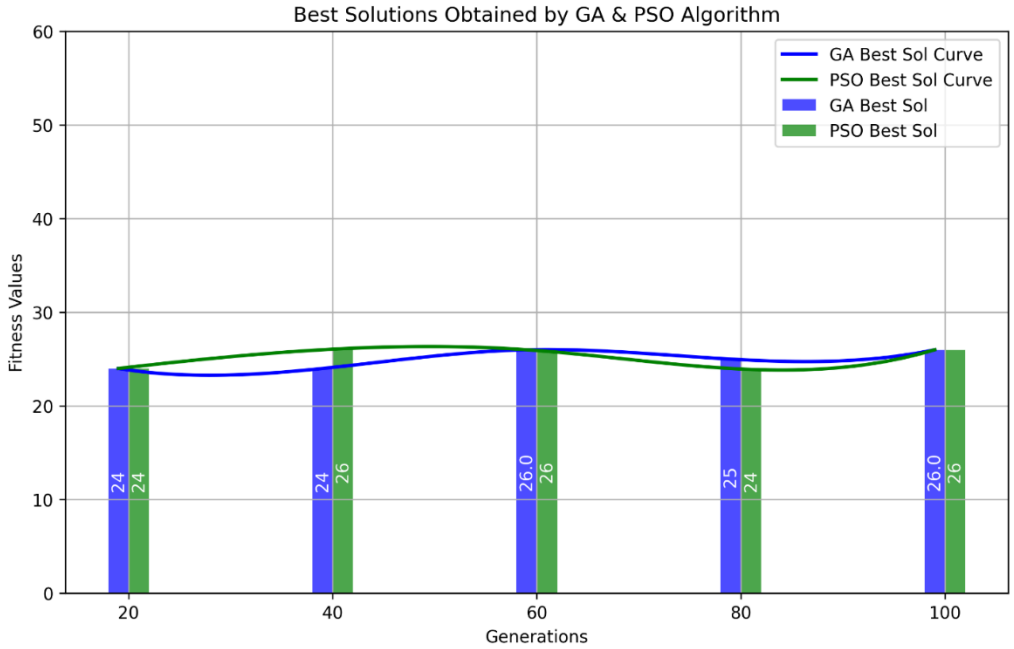

Figure 6.13 Poly la Dataset best solutions

The best solution values in (Figures 6.7 & 6.9) demonstrate the stability of the algorithms performance because they hold best solutions over several runs. The algorithms' stability shows that they have thoroughly investigated the solution space, as seen by the small variance in the best solutions over various generations. (Figure 6.8) on the other hand, shows a modest change in the curve when the number of generations is set to 100. This could be a sign of the algorithms slow convergence behavior, which suggests that further generations might enable a thorough investigation of the solution space.

In (Figure 6.10), the 20th and 40th generations were when ideal solutions were found, while the following generations saw the emergence of suboptimal solutions. This pattern shows that the algorithms may have difficulty maintaining diversity as they progress, even though they are successful in broad exploration in the early stages. (Figures 6.11-6.13) provide more evidence of the erratic nature of optimization, as shown by the variations in solution quality. The dynamics of optimization algorithms have inherent complexity that could explain this behavior.

In order to know when to stop or adjust the optimization process for better outcomes, careful monitoring of solution quality over generations is crucial.

Chapter 7

Conclusion

In this work, we studied, the problem of nesting with free rotation. The primary objective of the problem is to find the optimal placement of the parts within a specified range, with a key goal of minimizing the height at which parts are positioned. This helps in minimizing the overall space occupied by the parts.

The problem addressed in this work is two dimensional, with convex and non-convex parts which can rotate to any angle. The problem is governed by some ristrictions that must be adhered to in order for a proposed solution to be examine valid. Specifically, all parts must be alocated. Furthermore, all parts must remain within the fixed boundary of the range and there will be no overlapping of any part.

To answer the following question: “Is there any way to improve resolution efficiency? For the problem of nesting with continous rotation? We implement evolutionary algorithms, a Genetic algorithm (GA) and a Particle -Swarm-Optimization algorithm (PSO). To compose the proposed methods, two positioning rules, five sorting rules and two rotation rules given in [7] were implemented.

To represents the individual pieces, we use the form of polygons, whereby each piece is defined by an ordered arrangement of cartesian plane coordinates (x, y). The range is determined by a fixed constant, denoted by L, that represents its width along with a variable W, which represents the utilized height of the strip. The goal is to minimize the value of W.

To verify the overlapping issue of the pieces, an algorithm was proposed that transform the given pieces into a collection of triangles(triangulation of the pieces). By doing so, to check if two pieces are overlapping we just have to check if their respective triangles are overlap ?

To test the performance of the proposed methods, ten test instances of the literature [30] were used. All with varying quantities of pieces and convex pieces and not convex.

The Genetic algorithm is not able to run all the test instances which we get from [30]. This issue arise because some instances contained a higher quantity of parts with some more difficult shapes. For the test Instances that we use Genetic algorithm works well and complete its

execution with a little less time then PSO algorithm. For the PSO algorithm it works well with less number of generation but as we increase the limit of generations its execution face many issues like with a bigger generation value it takes very long time to execute a single generation and also in larger files it terminates without completing its full execution. We present our results in tabular form and than also plot some graphs to visualize the best solution found by both of the algorithms.

Bibliography

[1] H. H. H. S. Gerhard Wascher, “An improved typology of cutting and packing problems,” *European Journal of Operational Research,* vol. 183, pp. 1109-1130, 2007.

[2] J. F. O. Julia A. Bennell, “The geometry of nesting problems: A tutorial,” *European Journal of Operational Research,* vol. 184, pp. 397-415, 2006.

[3] L. R. Mundim, “Mathematical models and heuristic methods for nesting problems,” 2017.

[4] M. A. J. F. O. Jeinny Peralta, “SOLVING IRREGULAR STRIP PACKING PROBLEMS WITH FREE ROTATIONS USING SEPARATION LINES,” vol. 38, 2018.

[5] J. M. C. O. F. L. a. X. L. Xiaoping Liao, “Visual nesting system for irregular cutting-stock problem based on rubber band packing algorithm,” *Advances in Mechanical engineering,* vol. 8, pp. 1-15, 2016.

[6] A. P. a. T. R. Yuriy Stoyan, “Cutting and packing problems for irregular objects with continuous rotations: mathematical modelling and non-linear optimization,” *Journal of the Operational Research Society,* vol. 282, pp. 1-15, 2015.

[7] M. C. O. M. M. A. Wesley H. B. Nunes, “A Genetic Algorithm for the Nesting Problem With Continuous Rotations,” *IEEE Cobgress on Evolutionary Computation,* pp. 1107-1114, 2021.

[8] J. F. C. O. J. A. S. Ferreira, “Algorithms for Nesting Problems,” *Applied Simulated Annealing,* vol. 396, 1993.

[9] M. A. T. A. d. Q. Leandro R. Mundim, “A biased random key genetic algorithm for open dimension nesting problems using no-fit raster,” *Expert Systems with Applications,* vol. 81, pp. 358-371, 2017.

[10] D. R. Jones, “A fully general, exact algorithm for nesting irregular shapes,” *Journal of Global Optimization,* vol. 59, pp. 367-404, 2013.

[11] R. R. ,. A. M. G. F. M. T. ,. M. A. Pedro Rocha, “Two-Phase Approach to the Nesting problem with continuous rotations,” *IFAC-PapersOnline Journal,* vol. 48, pp. 501-506, 2015.

[12] ·. A. C. C. E. M. S. Luiz H. Cherri, “Mixed integer quadratically-constrained programming model to solve the irregular strip packing problem with continuous rotations,” *Journal of Global Optimization,* vol. 72, pp. 89-107, 2018.

[13] A. Elkeran, “A new approach for sheet nesting problem using guided cuckoo search and pairwise clustering,” *European Journal of Operational Research,* vol. 262, pp. 757-769, 2013.

[14] F. M. B. T. J. F. O. M. A. C. R. A.-V. Aline A. S. Leaoa, “Irregular packing problems: A review of mathematical models,” *European Journal of Operational Research,* vol. 282, pp. 803-822, 2019.

[15] R. A.-V. n, “A branch & bound algorithm for cutting and packing irregularly, shaped pieces,” *Int. J. Production Economics,* vol. 145, pp. 463-477, 2013.

[16] M. F. a. I. Luzzi, “Exact and Heuristic Methods for Nesting Problems,” 2004.

[17] D. Eberly, “Triangulation by Ear Clipping,” Geometric Tools , 2002. [Online]. Available: https://www.geometrictools.com/.

[18] S. V. b. W. B. D. Kathryn A. Dowsland a, “An algorithm for polygon placement using a bottom-left strategy,” *European Journal of Operational Research,* vol. 141, pp. 371-381, 2002.

[19] J. H. Holland, Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence, The MIT Press Cambridge,, 1992.

[20] C. K. R. M. I. I. a. S. D. P. LARRANAGA, “Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators,” *Academic Publishers,* vol. 13, pp. 129-170, 1999.

[21] J.-M. Y. Y.-F. T. C.-Y. K. H.-K. Tsai, “Some issues of designing genetic algorithms for traveling salesman problems,” *Soft Computing Publisher,* vol. 8, pp. 689-697, 2004.

[22] J.-M. Y. Y.-F. T. a. C.-Y. K. Huai-Kuang Tsai, “An Evolutionary Algorithm for Large Traveling Salesman Problems,” *IEEE Transaction on Systems, Man and Cybernetics Publisher,* vol. 34, pp. 1718-1729, 2004.

[23] J. I. H. a. R. P. R. Baraglia, “A Hybrid Heuristic for the Traveling Salesman Problem,” *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION,* vol. 5, pp. 613-622, 2001.

[24] B. R. a. P. Merz, “A Genetic Local Search Algorithm for Solving Symmetric and Asymmetric Traveling Salesman Problems,” in *Proceedings of IEEE International Conference on evolutionary Computation*, Indianapolis, USA, 1996.

[25] P. M. a. B. F’reisleben, “Genetic Local Search for the TSP: New Results,” in *International Conference on Evolutionary Computation,* Seigen, 1997.

[26] C. W. Reynolds, “Flocks, Herds, and Schools: A Distributed Behavioral Model,” in *Association for Computing Machinery*, New York, 1987.

[27] Y. S. a. R. Eberhart, “A Modified Particle Swarm Optimizer,” *IEEE world congress on*

computational intelligence, pp. 69-73, 1998.

[28] M. C. a. J. Kennedy, “The Particle Swarm—Explosion, Stability, and Convergence in a Multidimensional Complex Space,” *IEEE transactions on Evolutionary Computation,* vol. 6, pp. 58-73, 2002.

[29] A. A. H. &. A. K. A. Hwaita, “Movement Particle Swarm Optimization Algorithm,” *Modern Applied Science,* vol. 12, p. 148, 2018.

[30] “ESICUP Cutting and Packing,” Euro online, [Online]. Available: https://www.euro-online.org/websites/esicup/data-sets/.

[31] F.-A. F. M.-A. G. M. P. a. C. G. François-Michel De Rainville, “DEAP: A Python Framework for Evolutionary Algorithms,” *Genetic Programming and Evolvable Machines,* vol. 20, pp. 139-142, 2012.

[32] A. J. N. Juan J. Durillo, “jMetal: A Java framework for multi-objective optimization,” *Advances in Engineering Software,* vol. 42, pp. 760-771, 2011.

[33] F.-M. D. R. M.-A. G. M. P. C. G. Felix-Antoine Fortin, “DEAP: Evolutionary Algorithms Made Easy,” *Journal of Machine Learning Research,* vol. 13, pp. 2171-2175, 2012.

[34] S. Y. Jinhan Kim, “Software review: DEAP (Distributed Evolutionary Algorithm in Python) library,” *Genetic Programming and Evolvable Machines,* vol. 20, pp. 139-141, 2018.

[35] D. WHITLEY, “A genetic algorithm tutorial,” *Stat&tics and Computing,* vol. 4, pp. 65-85, 1994.

[36] “Creating Types,” [Online]. Available: https://deap.readthedocs.io/en/master/tutorials/basic/part1.html.

[37] D. B. D. R. a. M. R. R. Beasley, “ An overview of genetic algorithms,” *University Computing,* vol. 15, pp. 56-69, 1993.

[38] S. ·. S.N.Deepa, Introduction to Genetic Algorithms, Springer Berlin Heidelberg, 2008.

[39] D. T. Baeck, Evolutionary Algorithm in Theory and Practice, Oxford University Press , 1996.

[40] “Crossover (genetic algorithm),” [Online]. Available: https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm).

[41] E. S. a. M. Semenkina, “Self-configuring Genetic Algorithm with Modified Uniform Crossover Operator,” *Advances in Swarm Intelligence,* vol. 7331, pp. 414-421, 2012.

[42] S. Mirjalili, Evolutionary Algorithms and Neural Networks Theory and Applications, Springer international publishing AG part of springer nature , 2019.

[43] “Evolutionary Tools,” [Online]. Available: https://deap.readthedocs.io/en/master/api/tools.html#deap.tools.mutShuffleIndexes.

[44] “Go mutation,” [Online]. Available: https://pkg.go.dev/github.com/sineatos/deag/tools/mutation.

[45] “Evolutionary Tools,” [Online]. Available:

https://deap.readthedocs.io/en/master/api/tools.html#deap.tools.mutFlipBit.

[46] "Go Mutation," [Online]. Available: https://pkg.go.dev/github.com/sineatos/deag/tools/mutation#MutUniformInt.

[47] "Evolutionary Tools," [Online]. Available: https://deap.readthedocs.io/en/master/api/tools.html#selection.

[48] "Evolutionary Tools," [Online]. Available: https://deap.readthedocs.io/en/master/api/tools.html#deap.tools.selBest.

[49] "Evolutionary Tools," DEAP Documentation, [Online]. Available: https://deap.readthedocs.io/en/master/api/tools.html#deap.tools.selTournament.

[50] "Evolutionary Tools," [Online]. Available: https://deap.readthedocs.io/en/master/api/tools.html#deap.tools.selRandom.

[51] B. &. R. P. &. J. Kennedy, "Particle swarm optimization An overview," *Swarm Intell,* vol. 1, pp. 33-57, 2007.

[52] A. E. &. S. A. &. R. Dony, "Strength Pareto Particle Swarm Optimization and Hybrid EA-PSO for Multi-Objective Optimization," *Evolutionary Computation,* vol. 18, pp. 127-156, 2010.

[53] w. Y. S. James Kennedy and Russell C. Eberhart, Swarm Intelligence, Scott Norton, 2001.

[54] M. V. K.E. PARSOPOULOS, "Initializing the Particle Swarm Optimizer Using the Nonlinear Simplex Method," *Computer Engineering and Application Publisher,* vol. 43, pp. 47-48, 2002.

[55] J. Z. Y. L. a. Y.-H. S. Zhi-Hui Zhan, "Orthogonal Learning Particle Swarm Optimization," *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, vol. 15, pp. 1763-1764, 2011.

[56] "Particle Swarm Optimization Basics," [Online]. Available: https://github.com/DEAP/deap/blob/master/doc/examples/pso_basic.rst.

[57] "Particle Swarm Optimization Basics," [Online]. Available: https://deap.readthedocs.io/en/master/examples/pso_basic.html.