# Heuristic, meta-heuristic, and hyper-heuristic approaches for 2 dimensional bin packing

Student Name: Martin Ramalingum
Supervisor Name: Iain Stewart
Submitted as part of the degree of BSc Computer Science to the
Board of Examiners in the Department of Computer Sciences, Durham University

**Abstract**—This research addressed the 2-dimensional bin packing problem with rectangular pieces, which has applications in shipbuilding, textiles, and the ceramics industry. Heuristic techniques have been developed to provide good-quality solutions for different subproblems associated with bin packing, including the assignment of pieces to bins and the placement of pieces within bins. Recent literature has proposed high level heuristics which produce better results than standard heuristics, but the field has attracted criticism for vagueness and poor scientific procedure.
The aims of this paper were to reproduce a hierarchical hyper heuristic technique, to achieve better results than low level heuristics alone. The goals were broadened to consider high level heuristics in general, and we produced a genetic algorithm hyper heuristic that was able to evolve sequences that performed equally as well as the best low level heuristics. However, the hyper heuristic struggled to find sequences which were strictly better than the low level heuristics. We concluded that hyper heuristics must be carefully fine tuned and utilise a range of effective low level heuristics to perform well, and other methods could be used instead to obtain more practical solutions to industrial challenges.

**Index Terms**—Evolutionary computation and genetic algorithms, heuristic methods, optimization, packing

## 1 INTRODUCTION

### 1.1 Problem background and importance

THE class of bin packing problems encompasses a wide range of NP-Hard combinatorial optimisation problems [1]. Generally, the aim of the problem is to assign pieces (also referred to as 'items') to bins (also referred to as 'objects'). Bin packing problems are closely related to cutting stock problems, in which the aim is to cut pieces out of sheets (analogous to bins). Thus, in early studies much of the information and techniques from bin packing problems were used in cutting stock problems, and vice versa. In bin packing the aim is to pack all the pieces to minimise the number of objects used, or if only one object is used to minimise the wasted area in the object. Similarly, for the cutting stock problem the aim is to minimise the wasted 'trim' remaining after the pieces have been cut out. Much early research into cutting stock problems was conducted by Kantorovich [2]. It should be noted that single-bin packing problems can be reduced to knapsack problems by assigning a value to pieces then trying to maximise the total value of pieces in the bin rather than just the number of pieces.

A large number of problem variants arose, encompassing:

- 1, 2, 3 or n dimensional bins and pieces
- Single or multiple bin solutions
- Fixed or variable bin size
- Specific shapes, regular shapes, or irregular shape pieces
- Rules concerning specific aspects of the problem formulation such as: rotating pieces, reflecting pieces, or performing guillotine cuts to reclaim unused space.

In response to this, Dyckhoff [3] established a typology to categorise both cutting stock and bin packing problems, in addition to knapsack and scheduling problems. This project is based solely on the 2VIC designation: 2-dimensional, assigning all pieces to a selection of objects, identical objects, and congruent shapes. In the improved typology constructed by Wascher et al [4], the problem tackled is: 2-dimensional, input value minimisation (where the input value is the number of bins used), weakly heterogenous assortment (the pieces are of the same shape but varying size), several identical large (relative to pieces) objects, with regular (rectangular) pieces.

The large variety of problem types can be partially attributed to the range of industrial applications for solutions to the problem, with extensive literature in the fields of Mathematics, Operations Research, and Computer Science. Okano [5] describes a 2-dimensional bin packing problem in the context of shipbuilding, Martinez-Sykora et al. [6] specifically developed an cutting stock algorithm for a ceramics company in Spain, and Hu et al. [7] proposed an algorithm to pack fabric into a rectangular shape, minimising the length of the rectangle. Identifying and producing good solutions to different formulations of bin packing problems has beneficial implications for a vast range of industrial processes.

Due to the difficulty of the problem, various online and offline heuristic algorithms have been developed to produce reasonable solutions. However, these heuristics balance solution quality against computation time and the efficacy of individual algorithms can be highly contextual to the problem. High level heuristics, including: metaheuristics, hyper heuristics, and hierarchical hyper heuristics aim to achieve better solution quality than single heuristics by guiding and tuning a particular heuristic or by using a clever combination low-level heuristics.

### 1.2 Project aims and deliverables

This project was inspired by a recent study by Guerriero and Saccomanno [8] investigating the performance of a dynamic hierarchical hyper heuristic, which reportedly *'outperforms the state-of-the-art approaches in terms of both solution quality and efficiency'*. Due to the history of metaheuristic methods lacking scientific rigour [9], this claim warranted further investigation. The initial aims of the project were:

- To experimentally verify claims that hierarchical hyper heuristics outperform low-level heuristics.
- To use a range of low-level heuristics to improve the performance of high-level heuristics.
- To use techniques learned from the L2 AI Search module to improve the searching of the solution space.

### 1.3 Contribution of the project

Originally, the deliverables of the project consisted of an implementation of the hierarchical hyper heuristic outlined by Guerriero and Saccomanno, adding more low-level heuristics to select from. However, some specifics of the implementation were unclear and we were unable to get a response from the authors for clarification. Therefore, the scope of the project changed and in this project we produced: an implementation of the bottom left placement heuristic using Python and NumPy, an implementation of selection heuristics based on studies by Terashima-Marin et al [10], and an original hyper heuristic using a genetic algorithm to evolve a hybrid selection heuristic approach. The main research question tackled was 'can a hyper heuristic be evolved to produce, on average, better quality solutions than low-level heuristics for the 2D bin packing problem?'. This was investigated by running both low-level heuristics and the hyper heuristic on a previously-unseen testing set.

Section 2 details relevant literature. Topics covered include: the variations of the bin packing problem and the specifics of the problem tackled in this project, low-level heuristic methods (including selection, placement, and filling heuristics), and high level heuristic methods (including metaheuristics and hyper heuristics). Finally different methods of evaluating fitness are discussed including fractional utilisation and bin count with recoverable areas.

In section 3, we detail the implementation of the Bottom-left placement heuristic and the worst case placement scenario, and present findings from initial optimisation experiments. The selection heuristics used are presented and the choice of selection heuristics is described in 3.2. The latter half of section 3 outlines the implementation of the hyper heuristic, the creation of problem instances, and the testing procedure for the implementation as a whole.

Sections 4 and 5 present the findings of the project and discuss the success of the approach taken. In section 6, we review the key contributions of the work and discuss future extensions of the project.

## 2 Related Work

### 2.1 Problem formulation

Bin packing problems exist in many different variations, with many formulations originating from specific industries. The factors that vary between problems include: the dimensionality of the problem, the shapes of the pieces assigned to the bins and the rules for placing pieces. While different studies tackle different iterations of the problem, well-performing algorithms or heuristic techniques from one investigation are often transferable to others. For example, Lopez-Camacho et al. demonstrate that the Djang and Fitch heuristic (implemented for the 1-dimensional case in [11]) can be modified to perform well in 2-dimensional regular and irregular cases [12].

A solution to the problem is an allocation of n pieces N identical rectangular bins, with the aim of minimising the number of bins used. The problem is defined formally as follows: [8]:

Let P = $\{p_1, p_2, ...p_n\}$ denote the set of 2-dimensional pieces, where each piece $p_k, k = 1, ...n$ is a polygon of area $a_k$.

Let B = $\{b_1, b_2, ...b_N\}$ denote the set of N identical rectangular bins, where each bin has a fixed height of H and a fixed width of W.

It is assumed that all n pieces can fit into the N bins. The following rules must also be obeyed in this variation of the problem:

- Rotation of pieces for placement is forbidden.
- Flipping (reflection) of pieces is forbidden.
- All pieces must be convex.
- Pieces cannot overlap with other pieces.
- Pieces must fit entirely within the bin.

In this project a simplified version of the regular polygon problem, restricted to rectangular pieces as demonstrated in [10], was used to investigate the performance of different heuristics. This version is "the most widely studied" [12] and as such there exist a large body of low-level heuristics tailored to specific cases of the problem.

### 2.2 Low level heuristics

Heuristics aim to produce approximate solutions to a problem within a reasonable time, and are therefore highly applicable to NP-Hard problems with vast solution spaces.

Low level heuristics (also known as operators in the context of hyper heuristics) are commonly grouped into placement, selection, and filling heuristics, each of which help to generate different solutions to the subproblems involved in bin packing.

#### 2.2.1 *Selection heuristics*

Selection heuristics are executed first and decide, given a piece or set of pieces, which bin to allocate the item(s) to. They can use features of both the pieces and bins to inform their decisions. The quality of the solution produced by the choice is dependent on the placement heuristic used, hence different properties need to be considered for different placement heuristics. Figure 3 demonstrates the geometric criteria used by the selection heuristics implemented in [8]. If the placement heuristic places pieces bottom-up in columns, for example, the unplaced pieces would be ranked in descending order of Horizontality-Down (the ratio between the width of the base, and the total width of the piece), Verticality-Left (the ratio between the height of the left vertical side - if it exists - and the total height of the piece), and area. As such, the ideal initial piece would be a right-angled triangle of large area, with the right-angle vertex at the origin.

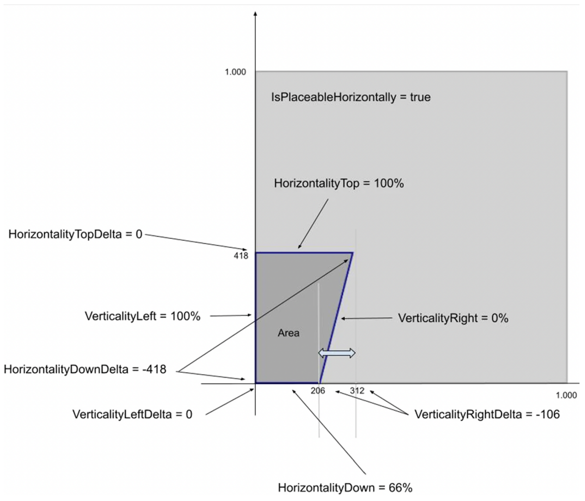


Fig. 1: Geometric properties considered by selection heuristics [8]

Other selection heuristics consider the properties of the bins. First-Fit Decreasing is a widely used selection heuristic that sorts the pieces in decreasing order of area, and creates a priority queue of bins. The heuristic iterates through the bins in the queue in decreasing order of priority, and places the piece in the first bin in which it fits. It is shown in [13] that in the 1-dimensional case this results in packings of $11M/9 + 4$ bins, where $M$ is the optimal number of bins. Other selection heuristics will execute placement heuristics on some or all of the bins, and will choose a bin based on the outcome. Best-fit considers all 'open bins' and places the piece in the bin such that the remaining area of the bin after placement is the minimum across all contenders. This solution quality/time tradeoff can be important to adjust when these heuristics are used by high level heuristics.

### 2.2.2 Placement heuristics

Placement heuristics describe how the piece should be placed, once allocated to a bin. Common placement heuristics include bottom-up placement, where pieces are placed bottom-to-top in columns of predetermined or variable width; left-to-right, where pieces are placed in rows of predetermined or variable height; and bottom-left, wherein pieces are initially inserted at the top-right corner of the bin, and are repeatedly moved down and left until a stable point is reached.

The latter algorithm has multiple versions to consider. Standard bottom-left placement involves inserting the piece such that it is fully within the bin, and must fit within the bin at each stage. Hopper and Turnton [14] present variations on this core functionality. Their version of the heuristic initially places the piece above the bin, and the piece is allowed to protude from the bin during each movement up until the final position is reached.

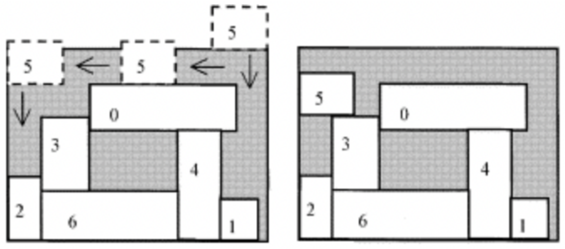


Fig. 2: Bottom-Left placement heuristic [14]

They also introduce the BLF (Bottom-Left Fill) algorithm as a hybrid variant that combines placement and filling heuristics, trading time complexity for efficacy. The piece is initially fitted at the lowest available valid position on the right hand side of the piece, and is then is moved to the leftmost valid position available. During the movement, the piece is allowed to collide with other pieces, provided that the final position is collision-free. While this filling operation can lead to higher-fitness solutions than standard placements, the fill variant has a time complexity of $O(n^3)$ [15] compared to the standard variant $O(n^2)$, making the BLF heuristic less desirable for use by hyper heuristics. As such, the standard variant was used in this project.

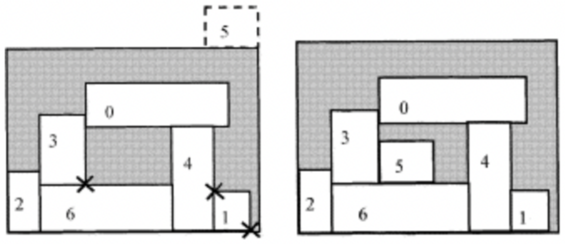


Fig. 3: Bottom-Left Fill placement heuristic. [14] piece 5 in put moved to the furthest down available space, then to the furthest left available space.

### 2.2.3 Filling heuristics

After the placement heuristic is unable to place any more pieces, filling heuristics can be executed to insert pieces into gaps in the structure (a common analogy describes the placement heuristic inserting the rocks, before the filling heuristic adds the sand). Guerriero and Saccomanno [8] describe the FA filling heuristic, which tries to optimise the position of the geometric centre of the added piece to be as close to the bottom-left of the bin as possible. FA first sorts the remaining pieces by descending area, then uses the Visual Vertex Search (VVS) algorithm, which considers placing the piece at each of the vertices formed by combined the previously-placed pieces into one structure. If vertices are too close together (according to predefined distances) they are not considered and if they are too far apart a virtual vertex is added. The heuristic then chooses the vertex position that minimises the barycentre of the piece, i.e. the furthest down and left.

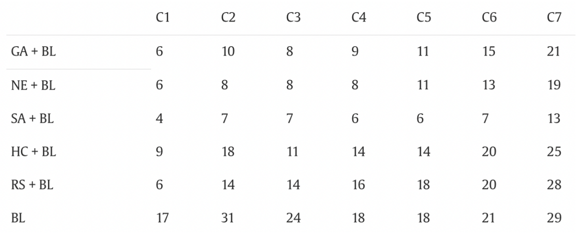
## 2.3 High level heuristics

The term 'High-level heuristics' encapsulates both metaheuristics and hyper heuristics. Metaheuristics "guide an underlying, more problem specific heuristic, to increase their performance" [16]. A common aim of metaheuristics is to improve upon local search methods for solving combinatorial optimisation problems by 'escaping' local optima. High level performance depends on the choice of low-level heuristics, as well as parameters associated with the exploration/exploitation tradeoff. The area of research is highly experimental and some publications have been criticised as being vague or having a poor theoretical basis [9]. However, both methods can provide solutions of reasonable quality for large search space where exhaustive searching is not feasible.

Hyper heuristics, as defined by Burke et al. are *"concerned with intelligently choosing the right heuristic or algorithm in a given situation"* [17]. Since low-level heuristics sacrifice solution quality for improvements in time, they often only work well in specific situations. The aim of hyper heuristics is to balance out the performance of different low-level heuristics (which can vary depending on the intricacies of the specific problem instance) to handle a variety of different problem instances.

While metaheuristics explore the space of solutions to try to find an optimal result, hyper heuristics explore the space of heuristics to try to find good heuristics for the problem at hand.

Hopper and Turnton [14] investigate the usage of metaheuristics for single-bin rectangle packing with rotations. These include: genetic algorithms, naive evolution, and simulated annealing. An initial population of 50 solutions for a given set of rectangles is created. These 50 are seeded individuals, the fittest out of a larger population of randomly-generated individuals. These solutions are encoded as a list of integers, each denoting a rectangle in the problem set, and individuals within the population consist of differently-ordered pieces. The authors discuss how sets of rectangles which are already sorted in order of descending height can lead to better quality packings [18]. The next generation is generated through the use of selection, crossover, and mutation operations. Proportional selection (also known as roulette-wheel selection) is used to choose potential parents, such that the probability of being chosen as a parent is proportional to the parent's fitness. Since duplicate values in an individual are not allowed (placing the same rectangle twice is forbidden), crossover and mutation operators must maintain the correct chromosome structure. Partially Matched Crossover (occurring with 60% probability) is used to interchange a consecutive subsequence of rectangles, and remove any duplicate values from the offspring. Multiple different forms of mutations are included, such as applying a 90° rotation with some low probability to every rectangle in the chromosome. Figure 4 shows the three metaheuristics consistently outperforming the low-level bottom-left placement heuristic across problem sets C1-C7. However, the difference in performance is larger in problem sets with fewer pieces and smaller objects (C1-C4) rather than more complex data sets (C5-C7). This improvement in solution quality also came at a cost: the time taken by the metaheuristics was consistently three orders of magnitude larger than the low-level heuristics, so the scalability of this solution to complex industrial problems is dubious.



| | C1 | C2 | C3 | C4 | C5 | C6 | C7 |
|---|---|---|---|---|---|---|---|
| GA + BL | 6 | 10 | 8 | 9 | 11 | 15 | 21 |
| NE + BL | 6 | 8 | 8 | 8 | 11 | 13 | 19 |
| SA + BL | 4 | 7 | 7 | 6 | 6 | 7 | 13 |
| HC + BL | 9 | 18 | 11 | 14 | 14 | 20 | 25 |
| RS + BL | 6 | 14 | 14 | 16 | 18 | 20 | 28 |
| BL | 17 | 31 | 24 | 18 | 18 | 21 | 29 |

Fig. 4: Relative distance between the best solution generated and the optimal height [14]

Terashima-Marin et al. [10] use a genetic algorithm as their hyper heuristic to select good selection and placement heuristics for both regular and irregular 2-dimensional problems. The hyper heuristic operates as demonstrated in Figure 5. Individual problem instances are labelled according to particular properties. For rectangle-piece bin packing, the label describes the distribution of pieces with different areas ('Huge', 'Large', 'Medium', 'Small'), and the height, ('Large', 'Medium', 'Small') and the fraction of pieces that have not yet been packed. Low level heuristics are applied to every problem in both the training and testing sets. For every problem in the training set, the low level heuristic which produces the highest fitness solution is saved. Each individual chromosome in the genetic algorithm consists of a variable number of 8-digit labelled blocks. When an individual is given a problem instance to solve, the low level heuristic is chosen by calculating the euclidean distance between the label of the problem and the label of all previously-encountered problems, and choosing the closest encountered problem. The corresponding best heuristics for the encountered problem are then applied to the new problem. The quality of the produced solution is used to assess the fitness of the individual, and selection, crossover, and mutation operations are applied to produce a high-fitness population. This procedure is run on the training set until some termination condition is met, by which time the hyper heuristic should have developed a varied 'library' of problem instances such that it is well suited to all kinds of pieces. The hyper heuristic is then run on the training and testing set to evaluate its performance.

## 2.4 Evaluating fitness

A naive approach to evaluating the success of a solution would be to count the number of bins used. However this method is insufficient to distinguish between solutions which use the same number of bins, which can occur frequently if the ratio of the total area of pieces $\sum_{k=1}^{n_i} a_k$, relative to the total area of the bins $\sum_{i=1}^{N} HW$ is small. This makes it difficult for high-level heuristics to choose effective low-level heuristics for a problem instance.

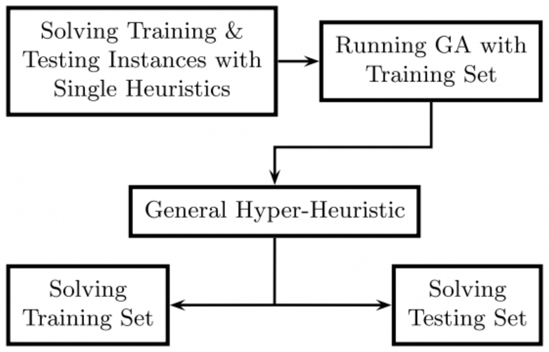

Solving Training & Testing Instances with Single Heuristics
Running GA with Training Set
General Hyper-Heuristic
Solving Training Set
Solving Testing Set

Fig. 5: Training and testing of the hyper heuristic. [10]

Therefore, two fitness metrics are commonly used when evaluating a solution. Terashima-Marin et al [10] utilise a fitness function which scores a solution between 0 and 1 depending on the number of bins used as well as the fractional utilisation of each bin. Fractional utilisation shown in equation 4 indicates how well a bin is packed by comparing the area of the pieces inside the bin to the area of the bin itself. Fitness is defined as the average fractional utilisation over all of the used bins, (equation 2).

$$U_i = \frac{\sum_{k=1}^{n_i} a_k}{HW} \tag{1}$$

where $n_i$ denotes the number of pieces in bin $i$.

$$F = \frac{\sum_{i=1}^{N} U_i^2}{N} \tag{2}$$

An alternative metric used by Han et al. [19] and Martinez et al. [6] is the K parameter, defined in equation 3. The parameter considers the number of bins used, similar to the naive approach, but replaces the final bin with the filled part of the bin that is obtained from performing a single guillotine cut across the bin. This is useful in contexts when the unused material of the bin can be reclaimed, such as in the ceramics sector. [6]

$$K = N - 1 + R^* \tag{3}$$

Where $R^*$ denotes the minimum filled area of the final (least-filled) bin that can be recovered by performing a horizontal or vertical guillotine cut.

## 3 METHODOLOGY

### 3.1 Placement heuristics

Individual bins were implemented as NumPy arrays of the corresponding size, i.e. W by H. Pieces were represented before the placement process as tuples containing a width and height value, both of which integers. The placement heuristic used in this project is a modified version of the standard bottom-left placement heuristic, as described in section 1.2.2 and detailed in Algorithm 1. The core functionality for running low-level heuristics is contained in *main.py* and the training procedure for the hyperheuristic is situated in *hyperheuristic.py*. The functionality described in section 3.1 can be found in *placement.py*.

#### 3.1.1 Placement procedure

This section describes the high level procedure for placing a piece into a bin, and section 3.1.2 outlines the process of individual movements and collision checking.

If the current bin is not empty, the piece is inserted into the top right corner. Until a collision occurs, the piece is repeatedly moved one place down. If a collision is detected, or the bottom of the piece passes the bottom of the bin, the piece is moved up one level.

Then, the piece is repeatedly moved left until a collision occurs or it passes the left side of the bin, in which case it is moved one place right. If the position of the piece (taken as position of the top right element in the piece submatrix) is the same before the round of downward and leftward movements, the movement procedure terminates.

The heuristic then returns the updated object with the piece added, the updated empty flag (set to 1), and a success flag set to 1, indicating successful placement. Otherwise, the movement is repeated until a stable position is reached.

When a piece is placed into a bin, the elements within the corresponding area of the bin matrix are incremented by a colour value, i. This value does not strictly alter the functionality of the placement, but describes the colour of the piece when plotting a graph of the packing. The colour value is generated differently depending on the selection heuristic used, but is always of the form: $i = (2x+1) \mod 10$, where x $\in \mathbb{Z}$. This maps values to the set {1,3,5,7,9}, which gives 5 distinct colour values for distinguishing between pieces and collision checking.

#### 3.1.2 Individual movement and collision checking

A single downward move consists of incrementing the matrix elements directly below the bottom row of the piece by i. If a collision is detected, this new row is decremented by i to restore the original values. Otherwise, the elements in the top row of the piece are set to 0, moving the piece one space down.

A similar process is applied for leftward movement, incrementing the elements in the column directly left of the piece by i and setting the rightmost column of the piece to 0. Collision checking is performed in the following cases:

- when the piece is first inserted into the bin (provided that the empty flag is set to 1),
- every time the piece is moved one space down,
- every time the piece is moved one space left.

When the piece is moved down, the elements in the bottom row of the piece are saved to the *row* variable. For every element $elem$ in *row*:

$$elem \leftarrow elem \mod 2 \tag{4}$$

If any element of the modified *row* is equal to 0, this indicates that a position in the matrix has been incremented twice (since i values are all odd). Thus, a collision has been detected. A similar operation is used for the column in leftward movement.

#### 3.1.3 Worst case bounds and optimisation

Let the width of the piece be $p_w$ and let the height be $p_h$. The theoretical worst case scenario is demonstrated in Figure 6,

**Algorithm 1** Bottom Left Placement

```
Input: (Object, empty flag), piece
if empty flag is None then
    Insert piece in bottom left corner
    empty flag $\leftarrow 1$
    success flag $\leftarrow 1$
    return updated object, empty flag, success flag
else
    Insert piece in top right corner
    if collisions occur along bottom row or left column of piece then
        success flag $\leftarrow 0$
        return original object, empty flag, success flag
    end if
    current position $\leftarrow$ top right corner of bin
    stable $\leftarrow$ False
    while not stable do
        previous position $\leftarrow$ current position
        while collision not detected and bottom row of piece is above bottom of bin do
            move piece 1 position down
            current position (y) $\leftarrow$ current position (y) - 1
        end while
        move piece 1 position up
        while collision not detected and left column of piece is right of left side of bin do
            move piece 1 position left
            current position (x) $\leftarrow$ current position(x) - 1
        end while
        move piece 1 position right
        if previous position = current position then
            break
        end if
    end while
    success flag $\leftarrow 1$
    return updated object, empty flag, success flag
end if
```

where unit square pieces are placed at $(W - p_w - 1, H)$ and $(W, H - p_h)$, and then in every position along the straight lines with gradient 1 that go through these points. In this case, $(H - p_h)p_w$ checks and $2(H - p_h)p_w$ increments or decrements must be performed for the downwards movement. Likewise, $(W - p_w)p_h$ checks and $2(W - p_w)p_h$ increments or decrements must be performed for the leftward movement.

Since the number of computations scales quickly with the dimensions of both the piece and bin, a number of optimisations were implemented to limit the time spent performing movements and checks.

Each bin in the queue is assigned an empty flag, initially set to 0 and then changed to 1 when a piece is placed in it. Instead of initially inserting a piece into the top right corner, the empty flag is first checked and if it is 0 then the piece is placed in the bottom left corner. In initial experiments, after the first piece had been placed, a 3-phase dynamic placement algorithm was used to reduce the number of movement and checking steps, as described in section 3.1.4. However, this method was only used in initial experiments as it was not compatible with certain selection heuristics and hyper heuristics.

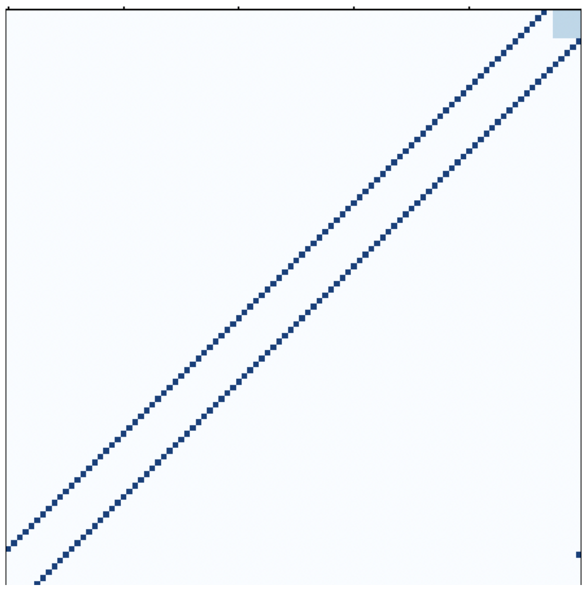


Fig. 6: Theoretical worst case for bottom-left placement, without modification. This requires $2(H - p_h)p_w + 2(W - p_w)p_h$ incrementing steps.

### 3.1.4 *Initial experiments*

After the first piece has been placed, phase 1 of the algorithm fills the bottom row of the bin with pieces. The placement heuristic checks if the width of the new piece is less than the remaining space between the right side of the bin and the rightmost piece. If this is the case, the piece is placed and the next piece is considered. Otherwise, phase 2 is initiated.

UR (denoting *'upperrightmost'*) denotes a theoretical point (shown as the red points in Figure 7) that divides the potentially filled portion of the bin from the portion that is known to be unfilled. The x coordinate of UR is that of the furthest right point of the rightmost shape, and the y coordinate is that of the highest row of the highest piece. These rightmost and uppermost points are shown as the green points in 7. During phase 2, the bottom left corner of the next piece is placed directly above and to the right of UR and the down-left placement loop is then executed. If the next piece cannot be placed at UR (i.e. UR is too close to the top or right corner of the bin), phase 3 is executed.

During phase 3 UR is fixed as the upperrightmost corner of the bin and the piece is inserted such that the top right corner of the piece is at UR. The standard placement loop is then executed. This phase is equivalent to the final placement heuristic that was used in the project.

## 3.2 Selection heuristics

The selection heuristics take as input the set of rectangular pieces to place as well as the bin dimensions. Each piece is then assigned to a bin, and placed using Bottom-left placement. We used commonly used selection heuristics from the literature including: First-fit, First-fit decreasing, Next-fit, Next-fit decreasing, Best-fit, Best-fit decreasing, and

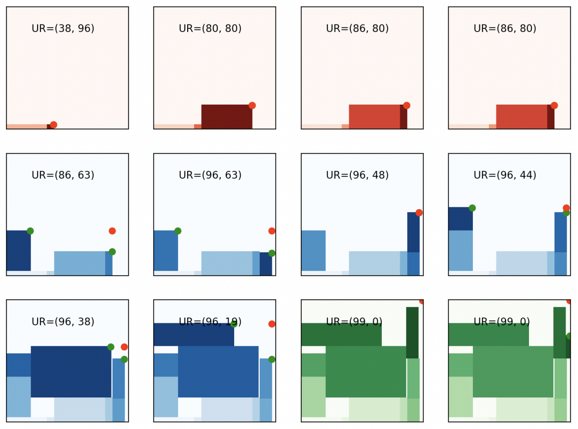


Fig. 7: Phase 1 (red): Filling bottom row. Phase 2 (blue): Placing piece at UR. Phase 3 (green): Standard filling from top right

the Djang and Fitch heuristic. The 'decreasing' algorithms operate similarly to their counterparts, but sort the pieces in decreasing order of area before executing the rest of the algorithm. All selection heuristics are implemented in *selection.py*.

The different selection heuristics are used by the hyper heuristic to generate a *'recipe'* for solving a problem [10], so differences between selection heuristics allow the hyper heuristic to adapt to a broad class of problems.

### 3.2.1 First-fit selection

First-fit (as mentioned in section 2.2.1 and detailed in algorithm 2) ranks the bins in a priority queue. The heuristic iterates through every piece in the list, trying to place the piece in the bin with highest priority. If this fails, the algorithm attempts to place the piece in the bin with the next highest priority. If the piece cannot be placed in any bin, a new bin is created and the piece is placed in it. The algorithm operates relatively quickly since no unnecessary placements are made. However, we hypothesised that the heuristic on its own would fare worse than other heuristics that compare multiple valid placements. This is because the solution produced is the first valid solution found by the algorithm with no guiding fitness function. First-fit decreasing was expected to outperform First-fit, since placing the largest pieces first should result in fewer gaps in the packing.

**Algorithm 2** First-Fit Selection

```
Input: pieces list, bin width, bin height
for each item do
    for each bin do
        bin, empty flag, success flag ← Bottom-Left((bin, empty flag), item)
        if success flag = 1 then
            update bin in memory and break
        end if
    end for
    if success flag = 0 then
        create new bin
        new bin, empty flag, success flag ← Bottom-Left((new bin, empty flag), item)
    end if
end for
update bins
```

### 3.2.2 Next-fit selection

Next-fit (Algorithm 3) operates similarly to First-fit, but maintains a 'current bin'. For each item, the heuristic attempts to place the piece in the current bin. If this fails, the next bin in the list is designated as the current bin and the algorithm attempts to place the piece in the new current bin. Next-fit differs from First-fit when a new piece is selected. Instead of placing the piece in the highest priority bin, the piece is instead placed in the current bin. If the piece cannot be placed in the original current bin or any of the subsequent bins in the list, the algorithm attempts to place the piece in the first bin in the list and attempts all bins thereafter until successful placement occurs or reaching the original current bin. If all bins have been unsuccessfully attempted, a new bin is created and the piece is placed in it. This new bin is then designated as the new current bin.

Next-fit was hypothesised to encounter fewer failed placement attempts than First-fit, since the high-priority bins fill up quickly in First-fit whereas pieces are likely to be distributed more even after Next-fit. Next-fit was therefore expected to run faster than First-fit, but produce solutions with lower fractional utilisation per bin.

**Algorithm 3** Next-Fit Selection

```
Input: pieces list, bin width, bin height
current bin pointer ← 0
success flag ← 0
for each item do
    current bin, empty flag, success flag ← Bottom-Left((current bin, empty flag), item)
    if success flag = 1 then
        update current bin in memory and continue
    else
        for o=1 to number of bins do
            current bin ← (current bin + o) mod (number of bins)
            current bin, empty flag, success flag ← Bottom-Left((current bin, empty flag), item)
            if success flag = 1 then
                update current bin in memory and break
            end if
        end for
        create new bin
        current bin ← new bin
        current bin, empty flag, success flag ← Bottom-Left((current bin, empty flag), item)
        update current bin in memory and break
    end if
end for
```

### 3.2.3 Best-fit selection

Best-fit (Algorithm 4) opts for a more computational expensive approach, trading time for solution quality. For each item, the heuristic attempts to place the piece in every bin. For each bin where the piece fits, the remaining area in the bin after placement is calculated. The bin with the least remaining area - and thus the greatest fractional utilisation - is chosen. While the use of a fitness function leads to higher quality solutions, the number of placement operations could be unsuitable for hyper heuristics with many iterations.

#### *3.2.4 Djang and Fitch selection*

The Djang and Fitch heuristic was designed for 1-dimensional bin packing, but was adapted to the 2-dimensional irregular-piece problem by Lopez-Camacho et al [12], in which both the size and shape of the piece have to be considered. The authors found that the heuristic performed well against established heuristics (including First-fit, Next-fit, and Best-fit) considering both time and solution quality (Figure 8).

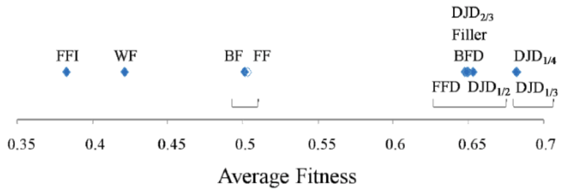


Fig. 8: Performance of the Djang and Fitch heuristics against other selection heuristics on 2-dimensional irregular-piece bin packing [12]. The fraction under DJD denotes the proportion of a bin that is filled before

The algorithm initially sorts all pieces by decreasing area. The *waste* value, denoting the amount of allowed unfilled space in the bin, is initialised to 0. In each round of placement, the current bin is filled with pieces until 1/3 of the area is filled. Three auxiliary algorithms are then used to check if 1 piece, 2 pieces, or 3 pieces can be added such that the unfilled space in the bin after placement is less than *waste*. If no piece was placed in the round and *waste* is less than the remaining unfilled area of the bin, the *waste* is incremented by some value $w$. Literature values recommend $w = HW \div 20$. Finally, if *waste* is greater than the free area of the bin and no pieces could be placed, a new bin is opened.

The implementation used in this project is described by Lopez-Camacho et al, with a minor modification to the auxiliary algorithms ( [12] Algorithms 3, 4). Originally, when placing multiple pieces, if the first piece was able to be placed but a later one could not be then the first piece was unplaced and a new first piece was tried. In this project, the first place was unplaced only after all subsequent second pieces were unable to fit. In addition two variants of the heuristic were used in the project, the first using the auxiliary algorithms to search for one extra piece, and the second searching for two.

DJT is a variant of the Djang and Fitch heuristic that considers placing up to 5 pieces every round instead of 3, and has been shown to have some success in 1-dimensional bin packing [20]. An implementation of DJT for 2-dimensional rectangle packing would be a useful topic for further investigation.

**Algorithm 4** Best-Fit Selection

```
Input: pieces list, bin width, bin height
for each item do
    for each bin do
        bin, empty flag, success flag ← Bottom-Left((bin, empty flag), item)
        if success flag = 1 then
            update bin in memory and break
        end if
    end for
    if success flag = 0 then
        create new bin
        new bin, empty flag, success flag ← Bottom-Left((new bin, empty flag), item)
    end if
end for
if pieces remaining to be packed then
    for each remaining item do
        waste ← bin height x bin width
        for each bin do
            bin, empty flag, success flag ← Bottom-Left((bin, empty flag), item)
            if success flag = 1 then
                if waste in bin < waste then
                    best bin ← bin
                    waste ← waste of current bin
                end if
            end if
        end for
        assign piece to best bin
        if success flag = 0 then
            create new bin
            new bin, empty flag, success flag ← Bottom-Left((new bin, empty flag), item)
        end if
    end for
end if
```

### 3.3 Generating and labelling problems

The creation of the training and testing sets was implemented in *training_and_testing_sets.py*. A problem set is defined as a fixed-length set of rectangles with:

$$1 \leq piecewidth \leq binwidth \tag{5}$$

$$1 \leq pieceheight \leq binheight. \tag{6}$$

Throughout this project the problem sets consisted of 20 rectangles and the bin dimensions were 100x100. The training set consisted of 1000 of these randomly-generated problems, and the testing set consisted of 500.

Each individual rectangle was produced by randomly generating a width and height value within the dimension ranges. The order of the problem set was then randomly shuffled. A random number of individuals within the problem set were then rotated; the length and height of the rectangle were swapped to perform a 90° rotation.

While these added random functionalities are not strictly necessary with random rectangle generation, they avoid 'easy' tessellation patterns which make the problem trivial if pieces are sorted by decreasing area. These patterns form when other common piece generation methods are used, such as in Figure 7 in [10].

Each problem was labelled using the criteria in [10] to describe the distribution of shapes and sizes of pieces, producing an 8-tuple with the following values:

1) The fraction of pieces with huge area: $p_h p_w > HW \div 2$
2) The fraction of pieces with large area: $HW \div 2 \geq p_h p_w > HW \div 3$
3) The fraction of pieces with medium area: $HW \div 3 \geq p_h p_w > HW \div 4$
4) The fraction of pieces with small area: $HW \div 4 \geq p_h p_w$
5) The fraction of pieces with tall height: $p_h > H \div 2$
6) The fraction of pieces with middle height: $H \div 2 \geq p_h > H \div 3$
7) The fraction of pieces with short height: $H \div 3 \geq p_h$
8) The fraction of remaining pieces relative to the original number of pieces in the problem set (initialised to 1).

All random functionality within the generation procedure, including: the width of the rectangles, the height of the rectangles, the extent of shuffling the rectangles, and the number of rotations was handled by the NumPy Random Generator class [21], where the seed for the Generator was given by the index in of the instance in the training set (in the range 0-999) or testing set (in the range 1000-1499). All random operations throughout the project - including in the hyper heuristic - were executed using this functionality for reproducibility.

## 3.4 Hyper heuristic implementation

### 3.4.1 Structure and Parameters

The intuition behind the hyper heuristic is that individual low-level heuristics may perform better in different conditions depending on the shapes and sizes of pieces in the problem set or the arrangement of the pieces in the bins midway through the packing process.

The hyper heuristic was implemented using a genetic algorithm [22] in which individuals (also referred to a chromosomes) within the population contain a sequence of the selection heuristics outlined in section 3.2. Each selection heuristic used is a modified version of the original heuristic. Rather than placing all of the items, each modified selection heuristic places only one and is given a label as shown in Table 1. An individual consists of a sequence of selection heuristic labels, describing the order the heuristics should be applied in. For example, in Table 2 the first piece is placed using Best-fit decreasing, the second piece is placed using Next-fit decreasing etc...

TABLE 1: Labels given to low-level heuristics used by the hyperheuristic

| **Label** | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| **Selection Heuristic** | FF | FFD | NF | NFD | BF | BFD | DJD |

TABLE 2: An example chromosome within the population, with a problem set of size 10

| **Chromosome** | 5 | 3 | 5 | 1 | 2 | 4 | 6 | 2 | 1 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|

For First-fit, Next-fit, and Best-fit the implementation of single-item placement was straightforward, as only the outer *'for each piece do'* loop had to be removed and extra some functionality to maintain a *current bin* variable had to be added.

Converting the Djang and Fitch heuristic proved more challenging since the main strength of the algorithm is its ability to find good combinations of pieces to place. The implementation used in the hyper heurisic was a compromise between keeping as much of the original functionality as possible while not letting the heuristic place multiple pieces per round. This is because every heuristic sequence involved would be dominated by Djang and Fitch placements and a convergence in scores would be likely to occur in the population. Therefore when a piece is placed, either in the initial filling when the bin is less than a third full or when a piece is added such that the free remaining space is less than the allowed waste, the algorithm immediately returns the updated bin and the hyper heuristic continues with the next low-level heuristic.

In every iteration of the genetic algorithm, a number of problems are randomly sampled from the training set and posed to each chromosome in the population. For each problem, the *problem-specific fitness* of individual chromosomes is assessed using the average fractional utilisation of each bin in the solution (Equation 2).

The *general fitness* of the chromosome was calculated by taking the median over the problem-specific fitnesses. This was chosen instead of the mean to avoid a high or low performance on a particular problem causing the general fitness to deviate greatly, as the hyper heuristic is more concerned with performing well over a broad range of problems rather than performing highly on a subset of problems but sacrificing solution quality in other subsets.

After a user-chosen number of generations, the genetic algorithm terminates and the individuals in the final generation are run on the testing set, for comparison against low-level heuristics.

### 3.4.2 Genetic operators

The parents of the next generation of individuals are decided by roulette wheel selection. Each individual is picked with a probability proportional to their general fitness.

After the parents are picked, they are paired up and two child chromosomes are produced by performing two-point crossover [23]. Two random indices in each of the chromosomes are chosen, and the first child inherits the sequence of the first parent up to Index 1, then inherits the sequence of the second parent between Index 1 and Index 2, and inherits the sequence of parent 1 after Index 2.

Elitist selection is also used to maintain good quality individuals within the population [24]. A proportion of the highest quality sequences are directly copied to the next generation without replacement or mutation. Elitist mutation is also used, wherein the elite group are also copied with a chance of mutation.

TABLE 3: Two point crossover. Information from Parent 1 is shown in red and information from Parent 2 is shown in blue.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Parent 1** | 5 | 3 | 5 | 1 | 2 | 4 | 6 | 2 | 1 | 3 |
| **Parent 2** | 0 | 2 | 3 | 2 | 4 | 5 | 3 | 3 | 2 | 0 |
| **Child 1** | 5 | 3 | 5 | 2 | 4 | 5 | 3 | 2 | 1 | 3 |
| **Child 2** | 0 | 2 | 3 | 1 | 2 | 4 | 6 | 3 | 2 | 0 |

Two forms of mutation can occur with small probability: point mutation and rotation mutation (Table 4). During a point mutation, a random position within the sequence is chosen and one heuristic is changed to a randomly-chosen other heuristic. A rotation mutation maintains the sequence but rearranges the order, with all heuristics being shifted a small, randomly-chosen number of places to the right. The parts of the sequence which are shifted past the end of the sequence are added at the beginning. Both forms of mutations aim to maintain the core structure of good sequences while performing small exploratory steps to search for better ones.

TABLE 4: A point mutation at index 6, and rotation mutation of size 3.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Original** | 5 | 3 | 5 | 1 | 2 | 4 | 6 | 2 | 1 | 3 |
| **Point Mutation** | 5 | 3 | 5 | 1 | 2 | 4 | 0 | 2 | 1 | 3 |
| **Rotation Mutation** | 2 | 1 | 3 | 5 | 3 | 5 | 1 | 2 | 4 | 6 |

### 3.4.3 *Experimentation and Issues*

As is common in metaheuristic and naturally-inspired search methods, the quality of solutions depends heavily on the parameters of high level heuristic. The performance of the hyper heuristic with parameter choices is also highly dependent on the problem, and the No Free Lunch theorem applies with regard to a large enough set of problems [25].

Some of the key parameters which affect the hyper heuristic are:

- **Population size**: a large population size can allow for greater exploratory searching, but quickly increases the number of computations required in each generation.
- **Number of problems per round**: having many problems allows for greater confidence in the quality of individuals. However in addition to the added computation costs, there is a risk of an early convergence of scores if the range of problems is too broad, due to the No Free Lunch Theorem. This would make the process of distinguishing between the quality of sequences difficult. A modifier that slowly increases the number of problems faced (similar to curriculum learning in machine learning [26]) may help to negate this problem.
- **Proportion of population chosen by elitist selection**: if too many individuals are chosen, the population will struggle to change and explore the search space. If the population is too small, good quality sequences may be lost.
- **Proportion of population chosen to reproduce**, and the proportion replaced by offspring: if too much of the population are chosen to reproduce, the number of computations will increase greatly and if too many members of the population are replaced by the new offspring, the population may quickly converge on different permutations of the same sequence. If too few individuals reproduce, the exploratory capability of the algorithm is severely limited.
- **Probability of mutation**: if mutations occur frequently, the algorithm will struggle to converge on high-quality sequences. If the probability of mutation is too small, the algorithm may be stuck in a local optimum with regard to general fitness.

After too many generations, the genetic algorithm frequently converged on sequences with many 1's and 5's with some 3's, corresponding to First-fit decreasing, Best-fit decreasing, and Next-fit decreasing, respectively. The first two algorithms performed well compared to other low-level heuristics, so it is likely that they were dominant in the population of the hyper heuristic. While these performed well relative to the low-level heuristics, it was difficult to perform exploratory steps due to the decline in diversity throughout the population.

## 3.5 Testing and validation

The hyper heuristic implementation had many dependencies in other modules, such as the selection heuristic functions and placement heuristic implementation. Therefore, unit testing was conducted throughout the project to verify that individual components were working correctly and to ensure that errors were not tracking throughout the programme. Unit testing was used over full system testing, since the hyper heuristic contains many random elements which are difficult to log.

For the placement functionality, the programme was tested with normal, boundary, and error cases. Normal cases consisted of small rectangles in empty or lightly-filled bins, and when a new bin had to be opened. Boundary cases featured large rectangles, rectangles which required multiple rounds of downward or leftward movement, arrangements in which rectangles were one unit too large or too small to fit into a gap, and cases where a new bin had to be opened despite there being free spaces in the current bin (due to the position of UR). Error cases consisted of rectangles with a height or width larger than the dimensions of the bin.

For selection heuristics, test examples were based on examples from literature or taking from hand-calculated problems.

For the hyper heuristic single-piece selection heuristic functions, testing was automated by initialising a chromosome consisting of only one selection heuristic and comparing the result to that of the low-level counterpart, which acted as an oracle.

## 4 Results

This section presents the findings from experiments conducted on the heuristics used throughout the project. The main aims of the experiments were to investigate the solution quality and time taken by the low-level and high-level heuristics.

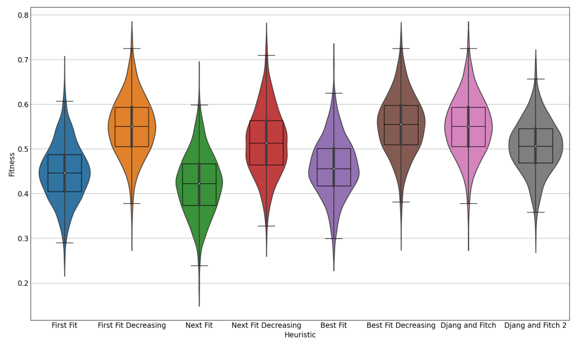
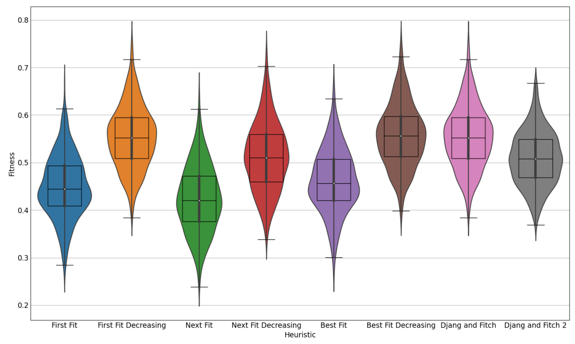


Fig. 9: Distribution of the fitness scores of low-level heuristics on the training set (above) and testing set (below).



The fitness scores of solutions were measured using fractional utilisation (equation 4). Time measurements of processes were taken by taking the difference between two calls - before and after the process - of the time.time_ns() function, which measures the integer number of nanoseconds since the epoch [27].

## 4.1 Low-level heuristics

Each of the selection heuristics were run on both the training set and testing set, with the latter set allowing for comparison against sequences produced by hyper heuristic.

Heuristics that performed decreasing area sort before placing pieces (the algorithms labelled 'decreasing' as well as Djang and Fitch) consistently outperformed those that did not perform sorting. The median (as well as most of the upper and low quartile) fitness scores for the heuristics were consistently between 0.4 and 0.6 (Figure 9).

The scores obtained follow trends seen in literature, as seen in Figure 8. Yet the relatively poor performance of the Djang and Fitch 2 heuristic was surprising. Lopez-Camacho et al. [12] used the 3-piece variant of the heuristic which may have been more successful. However in our experiments, giving the heuristic the power to place *more* pieces resulted in *lower* fitness (comparing Djang Fitch 1 and 2). In both the training and testing sets, the mean fitness for each heuristic was almost always equal to the median fitness (to a precision of 2 d.p.), suggesting that the data sets were large enough to prevent outliers skewing the mean.

Figure 10 shows the range of fitnesses produced after running the low-level heuristics on the first 100 problems in the training set. Some problems (45, 55, 63, and 96) led to a steep decline in scores as none of the algorithms were able to provide a better packing. Problems: 10, 12, 16 and 61 saw the majority of algorithms converge with higher scores. These may have been instances where the problems were almost fully sorted in order of decreasing area, so the more successful algorithms performed the same as their non-sorting variants.

Problems with converging scores were predicted to cause issues in the training of the hyper heuristic, providing less selection pressure to distinguish between good and bad parents. As a result, the elitist strategy may not maintain previously effective sequences between generations.

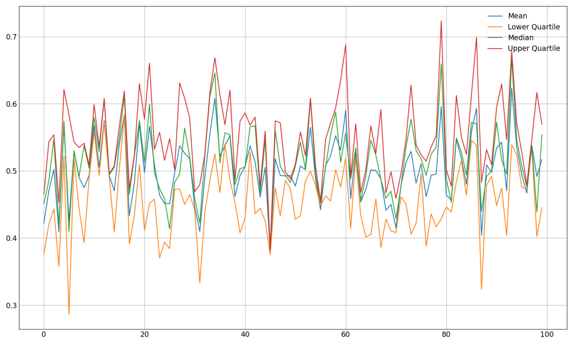


Fig. 10: The mean (blue), lower quartile (orange), median (green), and upper quartile (red) of fitness scores for the first 100 problems in the training set.

Table 5 shows the number of times that each selection heuristic produced the best solution. The trends seen in Figure 9 are also seen in the table: heuristics that sorted by decreasing area outperformed their counterparts, and Best-fit variants performed better than First-fit variants, which in turn outperformed Next-fit variants.

TABLE 5: The number of times each low-level heuristic performed the best on a problem set.

| **Heuristic** | **Training Set** | **Testing Set** |
|---|---|---|
| Best Fit Decreasing | 608 | 312 |
| First Fit Decreasing | 217 | 106 |
| Djang and Fitch 2 | 93 | 43 |
| Next Fit Decreasing | 62 | 23 |
| Best Fit | 12 | 11 |
| First Fit | 6 | 5 |
| Next Fit | 2 | 0 |
| Djang and Fitch 1 | 0 | 0 |

However, the weak performance of the Djang and Fitch 1 piece variant was surprising. It seems that the Djang and Fitch heuristic 1 piece variant performed well in situations where Best-Fit Decreasing also performed well, as evidenced in Figure 9. As the Djang and Fitch heuristic fills the current bin up until it is 1/3 full before the waste parameter is considered (section 8), some initial placements may have been sub-optimal. Therefore, it is likely that the Best-Fit heuristic often marginally outperformed Djang and Fitch, as evidenced by the greater median and upper and lower quartile scores in Figure 9. It was for this reason that rank based parent selection was not chosen in section 3.4.2, as this would have disproportionately reduced the chance of hyper heuristic sequences using Djang and Fitch 1 moves, compared to Best-Fit.

Tables 6 and 7 show the time taken (in milliseconds) by the selection heuristics on the training and testing problems,

TABLE 6

| | FF | FFD | NF | NFD | BF | BFD | DJD1 | DJD2 |
|---|---|---|---|---|---|---|---|---|
| mean | 7.19 | 6.44 | 7.11 | 6.38 | 14.36 | 17.63 | 27.86 | 88.19 |
| std | 0.81 | 0.57 | 0.65 | 0.67 | 3.42 | 5.11 | 3.6 | 30.6 |
| min | 5.41 | 4.87 | 5.3 | 4.69 | 7.76 | 8.19 | 17.97 | 21.32 |
| 25% | 6.76 | 6.04 | 6.66 | 5.89 | 11.98 | 13.8 | 25.5 | 67.48 |
| 50% | 7.13 | 6.41 | 7.08 | 6.33 | 13.94 | 16.85 | 27.55 | 84.21 |
| 75% | 7.56 | 6.83 | 7.52 | 6.83 | 16.26 | 20.67 | 29.79 | 104.62 |
| max | 23.66 | 8.68 | 9.54 | 9 | 33.67 | 50.32 | 60.49 | 277.67 |

TABLE 7

| | FF | FFD | NF | NFD | BF | BFD | DJD1 | DJD2 |
|---|---|---|---|---|---|---|---|---|
| mean | 6.95 | 6.45 | 6.95 | 6.28 | 14.06 | 17.2 | 27.46 | 86.41 |
| std | 0.59 | 2.52 | 0.59 | 0.63 | 3.1 | 4.58 | 3.3 | 28.75 |
| min | 5.63 | 4.67 | 5.16 | 4.48 | 7.37 | 8.97 | 18.29 | 26.06 |
| 25% | 6.54 | 5.99 | 6.55 | 5.85 | 11.77 | 14.07 | 25.22 | 66.83 |
| 50% | 6.9 | 6.32 | 6.92 | 6.24 | 13.61 | 16.37 | 27.31 | 82.98 |
| 75% | 7.29 | 6.66 | 7.32 | 6.67 | 16.1 | 19.46 | 29.5 | 101.74 |
| max | 8.8 | 61.35 | 8.9 | 8.39 | 23.97 | 37.51 | 47.71 | 223.34 |

respectively. As expected, heuristics that compared different potential solutions (i.e. Best Fit and Djang and Fitch) took much longer than the other heuristics. While in the scope of this investigation the time increase is not an issue, this may need to be considered for more complex implementations with larger bins and more pieces.

The predictions made in section 3.2.2 that Next-fit sacrificed the quality of solutions for increased speed compared to First-fit were somewhat correct, however the time improvement was negligible.

### 4.2 Hyper Heuristics

As outlined in section 3.4.3, there are many parameters that can affect the sequences evolved by the genetic algorithm. As a result, obtaining high performing sequences in the population required fine-tuning these parameters. Even with highly tuned parameters, the genetic algorithm struggled to outperformed the low-level heuristics used.

An early run of the genetic algorithms produced a population with a low diversity of sequences (Table 8), using the following parameters:

- population size = 20
- problems per round = 5
- elite proportion of population = 0.1
- mutated elite proportion of population = 0.1
- reproducing proportion of population = 0.8

As expected from the performance of the low-level heuristics, the highest scoring solutions consisted heavily of Best-fit decreasing moves as well as First-fit decreasing moves.

Throughout the testing process, the hyper heuristic would either converge on a similar sequence consisting of the best selection heuristics (referred to as ’converged sequences’) or would not converge, with sequences changing too much between generations to maintain useful subsequences (referred to as ’broad sequences’). Figure 11 shows the relative performance of the converged sequence (label 10 in Table 8) and a broad sequence, obtained with a population size of 40, 200 iterations, and 20 problems per round.

The converged hyper heuristic performed as expected, with the median fitness being on par with Best-fit decreasing, and the interquartile range being between that of Best-fit decreasing and First-fit decreasing. The broad sequence

TABLE 8: Converging population produced after 400 generations, with general fitness on the sample of 5 training set problems

| Label | Sequence Evolved | Fitness |
|---|---|---|
| 10 | [1, 5, 5, 5, 5, 1, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5] | 0.58 |
| 3 | [1, 5, 5, 1, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5] | 0.58 |
| 4 | [1, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5] | 0.58 |
| 6 | [1, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5] | 0.58 |
| 16 | [1, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 3, 5, 5, 5] | 0.58 |
| 8 | [1, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 3, 5, 5, 5] | 0.58 |
| 14 | [1, 5, 5, 1, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5] | 0.58 |
| 11 | [1, 5, 5, 1, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5] | 0.58 |
| 13 | [1, 5, 5, 5, 5, 5, 5, 5, 5, 5, 0, 5, 5, 5, 5, 5, 5, 5, 5, 5] | 0.58 |
| 17 | [5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 1, 3, 5, 5, 5, 5, 5, 0] | 0.58 |
| 5 | [1, 5, 5, 5, 5, 5, 5, 5, 5, 5, 0, 5, 5, 5, 5, 5, 5, 5, 5, 5] | 0.58 |
| 15 | [1, 5, 5, 5, 5, 5, 5, 5, 5, 5, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5] | 0.58 |
| 12 | [1, 5, 5, 1, 5, 5, 5, 5, 5, 5, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5] | 0.58 |
| 19 | [1, 5, 5, 5, 5, 5, 5, 5, 5, 3, 0, 5, 5, 5, 5, 5, 5, 5, 5, 5] | 0.54 |
| 1 | [1, 5, 5, 5, 5, 5, 5, 5, 5, 3, 0, 5, 5, 5, 5, 5, 5, 5, 5, 5] | 0.54 |
| 7 | [1, 5, 5, 5, 5, 5, 5, 5, 5, 3, 0, 5, 5, 5, 5, 5, 0, 1, 5, 5] | 0.54 |
| 2 | [1, 5, 5, 5, 5, 5, 5, 5, 5, 3, 0, 5, 5, 5, 5, 5, 0, 1, 5, 5] | 0.54 |
| 18 | [1, 5, 5, 5, 5, 5, 5, 5, 5, 3, 0, 5, 5, 5, 5, 5, 0, 1, 5, 5] | 0.54 |
| 0 | [1, 5, 5, 5, 5, 5, 5, 5, 5, 3, 0, 5, 5, 5, 5, 5, 0, 1, 5, 5] | 0.54 |
| 9 | [1, 5, 5, 5, 5, 5, 5, 5, 5, 3, 5, 5, 5, 5, 5, 5, 3, 5, 5, 5] | 0.54 |

was weaker than both Best-fit decreasing and First-fit decreasing, and the distribution of fitnesses was less even than the high performing heuristics, with the modal fitness (given by the width of the violin plot) being situated below the median. However, both hyper heuristics outperformed: the non-sorting algorithms, Next-fit decreasing, and Djang and Fitch 2.

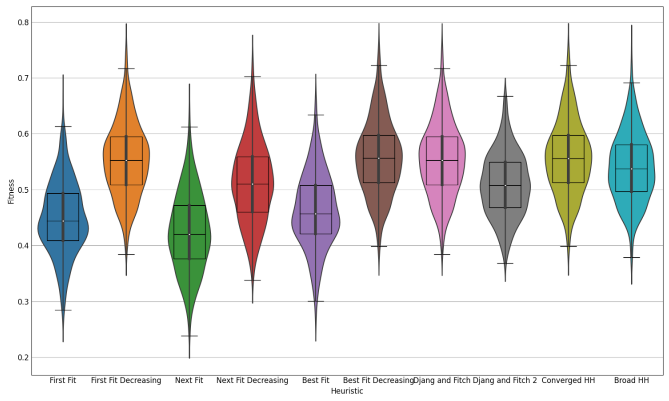


Fig. 11: Performance of a converged hyper heuristic and broad hyper heuristic relative to the low-level heuristics, when applied to the testing set.

## 5 Evaluation

In this section, we evaluate the outcomes of the project in relation to the deliverables and how they have provided insight into the research question: *’can a hyper heuristic be evolved to produce, on average, better quality solutions than low-level heuristics for the 2D bin packing problem?’*. The quality of the solution, as well as the software engineering and project organisation aspects of the projects are assessed.

## 5.1 Solution quality and choice of algorithms

Two of the original aims of the project have been confidently achieved. Firstly, the usage of low-level heuristics - in particular Best-fit decreasing and First-fit decreasing - improved the quality of sequences produced by the hyper heuristic. Secondly, the genetic algorithm structure and broad metaheuristic approach used, as well as the genetic operators (roulette wheel selection, elitist selection, crossover, and point mutation), were inspired by similar concepts used for the Travelling Salesman Problem in L2 AI Search. These concepts have been successfully applied in the context of bin packing.

The success of the hyper heuristic is somewhat mixed. On the one hand, it was successfully able to identify and converge on the strongest performing low-level heuristics, and produce solution fitnesses on par with Best-fit decreasing, First-fit decreasing, and Djang and Fitch 1. However, the hyper heuristic converged two quickly on a narrow subset of the selection heuristics available and was not able to perform useful exploratory moves (for example point mutations) to produce high quality solutions and escape local fitness maxima. To answer the research question, the hyper heuristic has produced better results, on average, than many of the hyper heuristics used, but only by heavily converging on the best heuristics available.

### 5.1.1 Performance of low-level heuristics

The initial assumption behind the hyper heuristic was that individual low-level heuristics are tailored to certain situations, and that under selection pressure from a wide range of problems the hyper heuristic would evolve sequences that apply each low-level heuristic to the situation that it is best suited for. In the case of 2-dimensional bin packing with rectangular pieces this does not seem to have been the case, as demonstrated in Figure 9 and Table 5. The most important factor in the fitness of a solution was the sorting of pieces beforehand.

The choice of heuristics was also not wholly suited to the problem. Next-fit based heuristics consisted performed poorly. While the decreasing variant achieved the best solution 85 times in the 1500 problems across the training and testing sets, the non-sorting variant had both the worst median and mean fitness out of all the algorithms used. The reason that First-fit had been regularly better-performing than Next-fit may have been because the initial bins in a solution are likely to be more packed. This would be the case for both algorithms, as the second bin is only opened after a piece cannot be placed in the original bin. Therefore, by attempting to place the current piece in the most packed bins first, the bins are each more likely to have a greater fractional utilisation and solution is likely to require fewer bins.

Refined First-fit [28] is an online algorithm that separates both the pieces and bins into classes, and pairs classes of pieces and bins such that pieces can be intelligently placed by getting the class of the piece and only assigning it to a bin of the corresponding class, even if there is empty space available in an opened bin. In 1-dimensional bin packing, it performs strictly better than both First-fit and First-fit decreasing. Therefore, replacing Next-fit with Refined First-fit may allow the hyper heuristic to reach greater local maxima. The use of the Djang and Fitch heuristic that places 5 pieces each round (DJT) may also be a useful addition, but it quickly increases the time complexity to potentially unfeasible levels for complex variants of the rectangular-piece bin packing problem (consider difference in time taken between DJD1 and DJD2 in Table 7).

### 5.1.2 Limits of the hyper heuristic approach

In addition to the problems caused by the low-level heuristics, other factors may have limited how well the hyper heuristic was able to produce stable sequences which could perform well on a variety of problems.

The single piece placement restriction for the selection heuristics used by the hyper heuristic may have negatively affected their performance. The Djang and Fitch heuristic struggled to survive and persist in sequences of the population. Since the Djang and Fitch heuristic often places multiple pieces per round (unlike the other heuristics used) it may have been disproportionately hindered by the limitation enforced.

Another reason that the hyper heuristic may have struggled is that if a sequence performed badly on a single problem set, it was unlikely to be replicated in the next generation. Therefore, the only solutions that survived were those that adhered strongly to the best low-level heuristics, since exploratory behaviour may have been punished in all but a few circumstances.

Terashima-Marin et al [10] used a more supervised learning approach by giving the hyper heuristic labelled problem sets, as described in section 2.3. Instead of using selection pressure alone to give the hyper heuristic feedback, individuals contain blocks consisting of: previously-encountered problems, their labels, and the best low-level heuristic found for the problem. In this method, the hyper heuristic has more agency to react to a novel problem as it can find a similar previously-seen problem using the label and apply the low-level heuristic associated. This model may have been useful for our problem, but it would still suffer from the same convergence problems due to the high performance of Best-fit. Considering on the training set, for example, their hyper heuristic would suggest applying Best-fit 608 out of the 1000 times.

Taking into account all of the problems experienced throughout the experiments: the disparate fitnesses of the low-level heuristics, the limitations of the hyper heuristic model, and the convergence of sequences produced by the genetic algorithm, a hyper heuristic may not be the approach to the problem.

Since many of the flaws of the implementation were a result of the low-level heuristics, a purely metaheuristic approach may have fared better. This is because metaheuristics choose between solutions, not other heuristics. The order that the pieces were placed in was a large indicator of the success of a solution, as evidenced by Figure 9 and the relative high performance of Refined First-fit [28]. Thus, a metaheuristic that used a single selection and a single placement heuristic and evaluated different permutations of pieces may have been more successful. For example, a simulated annealing algorithm that considers different sequence orders and uses Best-Fit selection and Bottom-left placement may be a successful method for the problem.

### 5.2 Software engineering and project management

Despite problems encountered over the course of the project, the deliverables were mostly achieved as a result of good software engineering and organisation principles.

We aimed to adhere to agile software development practices [29] throughout the course of the project by: regularly contacting the project supervisor to discuss objectives and problems, aiming to deliver software regularly, allowing aims and deliverables to change when needed, and frequently evaluating performance throughout the project.

The first half of the project revolved around researching hierarchical hyper heuristics and developing pseudocode to detail the methodology described in [8]. Due to the vague descriptions of the algorithms used, we attempted to contact the authors for clarification but received no response.

Instead of adhering to the Waterfall method of software development [30] and rigidly attempting to fulfill the initial deliverables of the project, we adapted the aims of the project to consider the field of high level heuristics more broadly, focusing on metaheuristics and hyper heuristics. This likely saved time by avoiding errors caused by trying to interpolate logic into Guerriero and Saccomanno's design without full insight into the inner workings of the algorithm.

The commitment to regular unit testing after each function or module was produced saved time over the course of the project. For example, if a bug caused by a boundary condition occurred undetected in the *placement.py* module, the selection heuristic functions in *selection.py* would have been affected and would have produced inaccurate results when called by the *main.py* module. Moreover, since the single-piece selection heuristic functions in the *hyperheuristic.py* module were checked against the original selection heuristic functions, errors would have persisted throughout the implementation.

The main problem encountered was that implementation was more time consuming than originally planned for, particularly in conjunction with other academic commitments. This hindered the regular delivery of code that had been envisaged. To mitigate the effects of this issue, the focus of the project temporarily shifted to researching other algorithms commonly used within the bin packing literature, including the Djang and Fitch heuristic.

However, this reduced the amount of time that could have been dedicated to conducting experiments and potentially adding new selection heuristics to the implementation. This problem should have been predicted when initially outlining the schedule of the project, and more implementation goals should have been assigned early on. Despite this, work was conducted flexibly and we adapted well to problems that arose.

In hindsight, the initial literature survey should have been conducted with greater critical judgement in order to assess both the quality of the heuristics proposed as well as the quality of the explanation. The choice to try to implement features from a relatively recent study conducted in 2022 [8] was risky and instead heuristic implementation with more peer-review should have been chosen, to prevent the initial problems that came about due to the lack of detail in the paper.

## 6 CONCLUSION

In this project we aimed to utilise commonly-used low and high level heuristics from literature sources to answer the research question: *'can a hyper heuristic be evolved to produce, on average, better quality solutions than low-level heuristics for the 2D bin packing problem?'*.

### 6.1 Contribution and key findings

We initially conducted a literature survey to investigate how heuristic, metaheuristic, and hyper heuristic methods have been used to produce good solutions for a range of bin packing problems. We also discuss the origins and applications of the problem including: industrial applications, typologies constructed to classify problems, and the similarities between bin packing and other problems such as cutting stock problems and the knapsack problem.

We developed an implementation of the 2-dimensional rectangular-piece bin packing problem in Python, and utilised the Bottom-left placement heuristic to place pieces into bins. A 3-phase optimisation technique was suggested for reducing the amount of time and computation required to place pieces and check for clashes. Although the optimisation was not used in the final hyper heuristic since the runtime was not a large problem, in larger scale versions of the problem (a common example used being 1000x1000-size bins with 60 pieces) it may be a useful addition to the solution.

Several selection heuristics, including: First-fit, First-fit decreasing, Next-fit, Next-fit decreasing, Best-fit, Best-fit decreasing, and the Djang and Fitch heuristic were implemented to assign pieces into bins. These selected heuristics were assessed using a training set of 1000 problems and testing set of 500 problems, being evaluated using a fitness function that calculated the mean fractional utilisation of the opened bins. Best-fit decreasing was found to outperform the other heuristics in most situations, although First-fit decreasing and the Djang and Fitch heuristic performed only marginally worse.

We then used the placement and selection heuristics to create a hyper heuristic, which aimed to choose the right low-level heuristic for each problem faced in order to develop a generic solution to a range of problem varieties. The hyper heuristic consisted of a sequence of selection heuristics that were the same length as the problem set. Each piece in the problem set was placed using the corresponding selection heuristic, until all items had been assigned to the bins. The aim of the hyper heuristic was to balance out the strengths and weaknesses of each selection heuristic to produce strong solutions in general.

The hyper heuristic was implemented with a genetic algorithm using genetic operators for: parent selection, crossover, and mutation, to develop the sequences of selection heuristic moves. The aim was to produce solutions with a greater average fitness than using low-level heuristics alone.

The hyper heuristic performed well relative to most of the low-level heuristics by converging on the best-performing heuristics (Best-fit decreasing and First-fit decreasing). However, despite experimentation with parameters the hyper heuristic only performed on par with the best low-level heuristic.

The core problems in the hyper heuristic stemmed from the uneven scores of the low-level heuristics, and that the initial assumption that different selection heuristics were suited to different situations was found to not hold. Best-fit decreasing was found to produce the best solution over in over 50% of problems, making it difficult for the genetic algorithm to enforce a selection pressure whilst maintaining enough genetic diversity to have a wide range of solutions within a population.

Overall, it seems likely that a hyper heuristic could be evolved to produce better quality solutions than a low-level heuristic. However, the structure of the hyper heuristic and the choice of low level heuristic must be considered carefully and an extensive experimentation procedure should be conducted to find the best parameters for the genetic algorithm.

Generally, hyper heuristics may not produce practical solutions to the industrial contexts mentioned due to the large-scale experimentation required for marginal improvements in results. In addition, some literature techniques may only work in specific or cherry-picked contexts and may not perform well broadly.

## 6.2 Future work

Due to the broad nature of the field of high level heuristics as well as the large variety with bin packing problems, there are many potential avenues for further investigation.

As previously mentioned, the order in which the pieces were assigned to bins was found to heavily affect solution quality. Therefore, a metaheuristic method such as simulated annealing that produces a permutation of pieces and uses a single selection heuristic and placement heuristic may be a more effective solution.

In addition, other low level heuristics could be used to attempt to escape the local maxima that the hyper heuristic found itself in. The use of other placement heuristics such as the Constructive Approach [31] in coordination with filling heuristics could create a variety of hybrid heuristic methods that perform well in different situations.

The same heuristics could also be applied to a range of 2-dimensional bin packing problems, for example:

- Bin packing with a placement heuristic that is allowed to rotate pieces.
- Bin packing with regular polygons instead of rectangles.
- Bin packing with irregular polygons instead of rectangles.
- Bin packing with variable bin dimensions.
- Bin packing with a fixed number of the bins, with the aim of packing as many items as possible or achieving the maximum fractional utilisation for each bin.

If hierarchical hyper heuristics become more prevalent in current bin packing literature, then an implementation of a hierarchical hyper heuristic that uses the placement and selection heuristics detailed in this report on a 2-dimension rectangular-piece bin packing problem would be a useful investigation. Findings from a similar investigation could then be applied to other 2-dimensional shapes (regular and irregular polygons). Furthermore, these could provide high quality solutions to other formulations of bin packing and cutting stock problems, similarly to how the Djang and Fitch heuristic was devised for 1D problems but proved successful in 2D [12].

## References

[1] A. A. Pandey, "An analysis of solutions to the 2d bin packing problem and additional complexities," *International Journal of Mathematics And its Applications*, vol. 9, no. 3, pp. 111–118, 2021.

[2] L. V. Kantorovich, "Mathematical methods of organizing and planning production," *Management science*, vol. 6, no. 4, pp. 366–422, 1960.

[3] H. Dyckhoff, "A typology of cutting and packing problems," *European Journal of Operational Research*, vol. 44, pp. 145–159, 01 1990.

[4] G. Wäscher, H. Haußner, and H. Schumann, "An improved typology of cutting and packing problems," *European journal of operational research*, vol. 183, no. 3, pp. 1109–1130, 2007.

[5] H. Okano, "A scanline-based algorithm for the 2d free-form bin packing problem," *Journal of the Operations Research Society of Japan*, vol. 45, no. 2, pp. 145–161, 2002.

[6] A. Martinez-Sykora, R. Alvarez-Valdés, J. A. Bennell, R. Ruiz, and J. M. Tamarit, "Matheuristics for the irregular bin packing problem with free rotations," *European Journal of Operational Research*, vol. 258, no. 2, pp. 440–455, 2017.

[7] X. Hu, J. Li, and J. Cui, "Greedy adaptive search: A new approach for large-scale irregular packing problems in the fabric industry," *IEEE Access*, vol. 8, pp. 91476–91487, 2020.

[8] F. Guerriero and F. P. Saccomanno, "A hierarchical hyper-heuristic for the bin packing problem," *Soft Computing*, pp. 1–14, 2022.

[9] K. Sörensen, "Metaheuristics—the metaphor exposed," *International Transactions in Operational Research*, vol. 22, no. 1, pp. 3–18, 2015.

[10] H. Terashima-Marín, P. Ross, C. Farías-Zárate, E. López-Camacho, and M. Valenzuela-Rendón, "Generalized hyper-heuristics for solving 2d regular and irregular packing problems," *Annals of Operations Research*, vol. 179, no. 1, pp. 369–392, 2010.

[11] P. Ross, S. Schulenburg, J. G. Marín-Blázquez, and E. Hart, "Hyper-heuristics: learning to combine simple heuristics in bin-packing problems," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pp. 942–948, 2002.

[12] E. López-Camacho, G. Ochoa, H. Terashima-Marín, and E. K. Burke, "An effective heuristic for the two-dimensional irregular bin packing problem," *Annals of Operations Research*, vol. 206, pp. 241–264, 2013.

[13] D. S. Johnson, *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.

[14] E. Hopper and B. C. Turton, "An empirical investigation of metaheuristic and heuristic algorithms for a 2d packing problem," *European Journal of Operational Research*, vol. 128, no. 1, pp. 34–57, 2001.

[15] B. Chazelle, "The bottomn-left bin-packing heuristic: An efficient implementation," *IEEE Transactions on Computers*, vol. 32, no. 08, pp. 697–707, 1983.

[16] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Comput. Surv.*, vol. 35, pp. 268–308, 01 2001.

[17] E. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, and S. Schulenburg, "Hyper-heuristics: An emerging direction in modern search technology," in *Handbook of metaheuristics*, pp. 457–474, Springer, 2003.

[18] E. G. Coffman, M. R. Garey, and D. S. Johnson, "Approximation algorithms for bin-packing—an updated survey," *Algorithm design for computer system design*, pp. 49–106, 1984.

[19] W. Han, J. A. Bennell, X. Zhao, and X. Song, "Construction heuristics for two-dimensional irregular shape bin packing with guillotine constraints," *European journal of operational research*, vol. 230, no. 3, pp. 495–504, 2013.

[20] N. Pillay, "A study of evolutionary algorithm selection hyper-heuristics for the one-dimensional bin packing problem," *SACJ*, vol. 48, pp. 31–40, 06 2012.

[21] "Random generator - numpy v1.24 manual." https://numpy.org/doc/stable/reference/random/generator.html numpy.random.Generator. Accessed: 2023-04-20.

[22] J. H. Holland, "Genetic algorithms," *Scientific american*, vol. 267, no. 1, pp. 66–73, 1992.

[23] G. Syswerda *et al.*, "Uniform crossover in genetic algorithms.," in *ICGA*, vol. 3, pp. 2–9, 1989.

[24] S. Baluja and R. Caruana, "Removing the genetics from the standard genetic algorithm," in *Machine Learning Proceedings 1995*, pp. 38–46, Elsevier, 1995.

[25] D. H. Wolpert, W. G. Macready, *et al.*, "No free lunch theorems for search," tech. rep., Citeseer, 1995.

[26] P. Soviany, R. T. Ionescu, P. Rota, and N. Sebe, "Curriculum learning: A survey," *CoRR*, vol. abs/2101.10382, 2021.

[27] "Time access and conversions - python 3.11.3 documentation." https://docs.python.org/3/library/time.htmlepoch, note = Accessed: 2023-04-20.

[28] A. C.-C. Yao, "New algorithms for bin packing," *J. ACM*, vol. 27, p. 207–227, apr 1980.

[29] K. L. Beck, M. A. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. J. Mellor, K. Schwaber, J. Sutherland, and D. A. Thomas, "Manifesto for agile software development," 2013.

[30] K. Petersen, C. Wohlin, and D. Baca, "The waterfall model in large-scale development," in *Product-Focused Software Process Improvement: 10th International Conference, PROFES 2009, Oulu, Finland, June 15-17, 2009. Proceedings 10*, pp. 386–400, Springer, 2009.

[31] M. Hifi and R. M'Hallah, "A hybrid algorithm for the two-dimensional layout problem: the cases of regular and irregular shapes," *International Transactions in Operational Research*, vol. 10, no. 3, pp. 195–216, 2003.