# Human-Evolutionary Problem Solving through Gamification of a Bin-Packing Problem

## ABSTRACT

Many complex real-world problems such as bin-packing are optimised using evolutionary computation (EC) techniques. Involving a human user during this process can avoid producing theoretically sound solutions that do not translate to the real world but slows down the process and introduces the problem of user fatigue. Gamification can alleviate user boredom, concentrate user attention, or make a complex problem easier to understand. This paper explores the use of gamification as a mechanism to extract problem-solving behaviour from human subjects through interaction with a gamified version of the bin-packing problem, with heuristics extracted by machine learning. The heuristics are then embedded into an evolutionary algorithm through the mutation operator to create a human-guided algorithm. Experimentation demonstrates that good human performers augment EA performance, but that poorer performers can be detrimental to it in certain circumstances. Overall, the introduction of human expertise is seen to benefit the algorithm.

## CCS CONCEPTS

**Computing methodologies** → Machine learning → Machine learning approaches → Bio-inspired approaches → Genetic algorithms;

**Applied computing** → Operations research → Decision analysis → Multi-criterion optimization and decision-making

## KEYWORDS

Business planning and operations research, Games, Heuristics, Interactive evolution, Machine learning

## 1 INTRODUCTION

There are many complex operational research problems arising from the areas of cutting and packing [1]. Problems with real world applications often requiring the use of optimisation techniques to solve. One such problem is bin-packing [2], which consists of a number of container objects (bins) and a fixed number of items that need to be stored in them (boxes). The bins are usually a large fixed size but can also vary, while the boxes are almost always an assortment of smaller sizes. The objective is to fit the boxes into as few bins as possible without violating the bin size constraints. The problem can have various dimensions and rises in complexity as the dimensionality increases.

Early attempts to solve the bin-packing problem examined several approximation algorithms, often based on very simple rules such as first fit (packing each box into the first bin it will fit into) [3]. Additional algorithms have been created based on heuristics derived from observation, analysis, or speculation, and the performance of these algorithms has been tested against the simple approximation algorithms by various studies [4, 5]. A branch-and-bound algorithm making use of some of these heuristics also proved effective at finding good approximations [6]. However, none of these approximation algorithms are guaranteed to provide an exact solution to an instance of the problem.

Evolutionary algorithms (EAs) are a tried and tested method for solving complex problems for which it is computationally infeasible to generate an exact solution. The generalisation of EAs allows them to be applied to many problems to generate good approximate solutions. They use simple automated processes requiring no human input after the initial encoding of the problem representation.

Due to the capabilities of EAs many attempts have been made to apply them to the bin-packing problem with various degrees of success. Several of these studies found that an EA by itself often performs poorly unless combined with other techniques. These include combining a grouping genetic algorithm with a local optimisation technique that obtained results superior to using either technique in isolation [7]. Another study used a biased random key genetic algorithm combined with some simple heuristics to obtain solutions to both 2D and 3D bin-packing problems [8]. Combining a genetic algorithm with a best fit decreasing approximation algorithm to avoid infeasible solutions [9] was also investigated.

Burke et al. [10] used genetic programming to create an effective algorithm to solve bin-packing problems, allowing for algorithms to be evolved based on the state of the bins. An interesting result of this study was that the best of the obtained evolved algorithms behaved almost identically to the first fit approximation algorithm. This demonstrated how useful heuristics can be derived from attempts to solve instances of the bin-packing problem. Combining automatically generated rule-based and data-based heuristics with a multi-objective optimisation problem was also found to be effective, though this was not applied to bin-packing [11].

Metaheuristics, such as EAs, are problem agnostic and good at reaching a goal but can often take a long period of time or require significant processing power to do so. Heuristics tend to be problem specific and rely on an understanding of the problem or the solution, or an approach that is known to be effective. Though heuristics can often offer quicker and easier ways of doing things, they might not always reach their goal.

Hyper-heuristics make use of a variety of metaheuristic and heuristic methods to try to take advantage of the benefits of both approaches. Hyper-heuristics have been used to generate heuristics that can be turned into readable algorithms [12] and have been applied to bin-packing with some success [13, 14]. Hyper-heuristics can encounter a couple of problems in their application, chiefly the extra resources required to decide which heuristic to make use of under what circumstances and providing

the hyper-heuristic with a full library of different heuristics to select from.

Effective heuristics can be derived from human approaches to solving a problem. This has been achieved with limited success through simple techniques that capture human behaviours to apply to robots [15], and to analyse the heuristics from human participants used in optimising routing problems [16].

Human-guided search has been investigated by Klau et al. [17], who applied it to a variety of optimization problems including a type of packing problem. Murawski and Bossaerts [18] investigated the heuristics used by participants presented with the knapsack problem, a problem of a similar nature to bin-packing. Murawski and Bossaerts were able to recognise a common human approach of applying a heuristic similar to the greedy algorithm followed by a heuristic similar to a branch-and-bound algorithm.

To best take advantage of human generated heuristics, it is important to understand that not all individuals are equally good at solving problems. Therefore, the best heuristics would presumably be generated by those with expertise or domain specific knowledge of the problem at hand. While this expertise could be assessed prior to trying to capture any heuristic the user applies to the problem, this could also be decided either during or after the process by scoring the user on their performance. This would involve giving the user feedback through a scoring system and an interactive visual representation of the problem, which would involve gamification of the bin-packing problem.

In applying gamification to the problem of linking gene patterns to predicted breast cancer outcomes, Good et al. [19] were able to make use of a crowd of both expert and non-expert users to test their hypothesis. Their game was able to capture useful knowledge from their expert players, which was then used to train a decision tree classifier. They also found that the players without domain specific knowledge performed less well. This was due to the representation of the problem needing to be kept complex for the experts to have a chance to take advantage of their expertise, making the game much harder to play for the non-experts.

To capture human derived heuristics a gamified version of the bin-packing problem is proposed here. This game captures the problem state and human input at each stage as the user solves a simple 2D version of the bin-packing problem. After the problem is solved, machine learning techniques are then applied to this data and the heuristics employed by the human user are derived. These derived heuristics are then used in place of or alongside of the mutation operator in an EA to determine if they improve the performance of the optimisation algorithm.

# 2 EXPERIMENTAL AND COMPUTATIONAL DETAILS

## 2.1 Problem Definition

For the purpose of this paper the bin-packing problem will be defined as follows. The problem consists of a fixed number of bins and exactly twice that number of boxes, the number of which determines the level of difficulty. Each bin has two dimensions, labelled as size and weight, the capacities of which are fixed and identical. The boxes have the same two dimensions, but their values are randomised. This is done in such a way that the sum of the weights and sizes of the boxes is enough to exactly fill half of the bins. The approach taken is to randomly generate the boxes by splitting half of the bins into slices and then shuffling and distributing them evenly between all the bins.

The objective is to minimise the number of bins being used, while the user interacts with the problem by selecting a single box from any bin and choosing which bin to move it to. The size and weight capacities of the bins act as constraints that can be temporarily violated to generate an infeasible solution. However, if a bin is already over-capacity in either dimension, no more boxes can be moved into it. The user is not allowed to submit an infeasible solution to be assessed and scored, and, due to the way in which the problem is generated, there is always a guaranteed optimal solution.

The scoring is calculated based on the number of full and empty bins, followed by the distribution of boxes between the partially-filled bins. This is to encourage the user to try to fill bins exactly while using as few as possible. The optimum score for a problem is calculated by multiplying the total number of bins by the sum of the maximum size and weight capacities

$$Score_{Max} = NoOfBins(Size_{Max} + Weight_{Max}). \quad (1)$$

All other scores are calculated by summing the individual totals for each bin, with the score per bin decided by a conditional statement. If the size and weight of the boxes contained in a bin is zero or equals both maximum capacities then the bin scores the sum of the maximum size and weight capacities. Otherwise, the bin score is calculated as the sum of the absolute difference from half the size capacity and half the weight capacity and then the bin scores are all summed to determine the problem score

$$Score = \sum_{Bin=1}^{n} \begin{cases} Size_{Max} + Weight_{Max}, if\ Bin_{cap} = Cap_{Max}\ or\ Bin_{cap} = 0 \\ \left|\frac{Size_{Max}}{2} - Bin_S\right| + \left|\frac{Weight_{Max}}{2} - Bin_W\right| \end{cases} \quad (2)$$

In these equations $Size_{Max}$ is the maximum size capacity of a bin and $Weight_{Max}$ is the maximum weight capacity of a bin. $Bin_S$ is the filled size of the current bin, $Bin_W$ is the filled weight of the current bin, $Bin_{cap}$ is the filled capacity of the current bin in both size and weight, $Cap_{Max}$ is the maximum capacity of the bin for either size or weight, and $n$ is the number of bins.

A value of 500 was decided upon for the bin size capacity based on the screen size of the object in pixels, and the bin weight capacity was set to match to keep the two dimensions equal. After a few trials the number of bins and boxes were set to 4 and 8 respectively for a problem that players solved easily (the easy problem), and 6 and 12 for a more difficult problem (the medium problem). A third, harder problem with 8 bins was also created but because of poor user performance on the easier two problems it was not taken further.

This version of the problem differs from many other implementations by not allowing new bins to be created, and by starting the problem with the boxes already distributed between the bins. This brings it closer to real world equivalents of the problem to allows users to employ their knowledge and expertise in solving it.

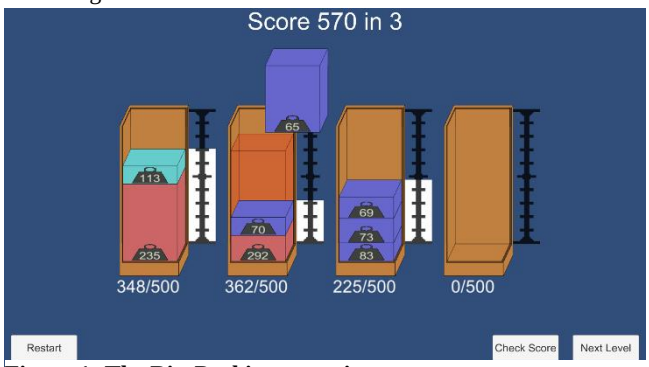


**Figure 1: The Bin-Packing game in progress**

## 2.2 Gamification and Implementation

Development of the bin-packing game was carried out using C# and the Unity Game Engine. The game screen consisted of a plain background with visual representations of the bins and boxes in an isometric view in the centre of the screen and a small number of user interface (UI) elements (**Fig. 1**).

A 'weight' symbol on each box showed the numerical weight value of that box. Additionally, as can be seen in **Figure 2** five colours were used to show the weight of the box relative to the minimum and maximum box weight values.

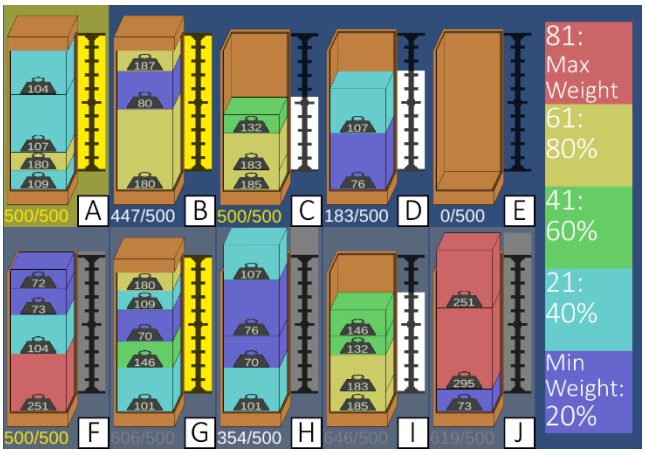


**Figure 2: All possible bin states during gameplay.**

Each bin displayed underneath itself the total current weight held by that bin as a numerical value out of the maximum bin weight capacity. The size of each box could only be judged by sight, as the screen size of each box in pixels directly related to the size value of that box. The bin size capacity was shown by an unmarked scale adjacent to the side of the bin with a white bar indicating fullness. Whenever a box was selected it would be removed from the bin it was in and a transparent 'ghost' image of the box would highlight how it would change the bin capacity of any bin the box hovered over as the user moved it around.

If a bin was exactly filled in size, a lid would appear on it (**Fig. 2**; **A**, **B**, **G**), while if it was exactly filled in weight the text underneath would turn yellow (**Fig. 2**; **A**, **C**, **F**); if both then the bin would also be surrounded by a yellow box (**Fig. 2; A**). Conversely, the bin would be surrounded by a grey box if it violated the constraints (**Fig. 2**; **F**, **G**, **H**, **I**, **J**).

If the size constraint was violated (i.e. the boxes in the bin had a total combined size greater than 500) then the scale to the right of the bin would turn grey (**Fig. 2**; **F**, **H**, **J**). If the weight constraint was violated then the text underneath the bin would turn grey (**Fig. 2**; **G**, **I**, **J**), and if both were violated then both would happen (**Fig. 2**; **J**). If the constraints of any bin in the game were violated then the solution was considered infeasible.

The user was told the optimum score before they played and encouraged to compete with other players by achieving it in the minimum number of moves. The game state and score at each move was then logged in a text file.

The game was demonstrated to prospective undergraduate students and their family members who were then encouraged to play it. Several individuals attempted the game, with a total of ten users playing and successfully completing the easy 4-bin game, three of which then also completed the medium 6-bin game.

## 2.3 Machine Learning

When deciding what to learn from the gathered data several decisions needed to be made, the first of which was how best to represent the problem. This needed to be carried out in a way that allowed any problem-solving heuristic captured from the data to be generalisable rather than only applicable to this specific problem instance. This also needed to be done in such a way that it took best advantage of the player capabilities.

Each move of a box could be broken down into two parts; target box selection followed by target bin selection. This could however be confused by composite moves, in which a box might be moved such that it temporarily makes the problem worse but overall allows the user to solve the problem more quickly and efficiently.

However, the easy 4-bin problem could be solved in as little as 6 or 7 moves which would make recognising composite moves difficult. This is also confounded by players moving boxes back and forth between the bins while deciding where to place them. Given this, it was felt best to only look at single moves in the current study.

The box selected could be decided at random and any heuristic would theoretically still apply. The opposite might not be true, so it was decided to use machine learning to capture which bin a chosen box would be put into rather than which box was selected.

In this initial experiment, only moves that improved the score were included in the dataset for training. This ignored bad moves made by players learning how to play the game or players who struggled, but still allowed any good move to aid the learning process. To generalise the problem representation only relative properties of the problem (rather than specifics) could be used for

learning, and the two dimensions were combined into a single total.

Several potential machine learning approaches were considered for this task, and the decision tree regressor was selected. The main reason was its ability to generate human-understandable models of the players' behaviour. This allows for the tree to be sense-checked to ensure that it has captured a reasonable approximation of human problem solving in this task. The sklearn decision tree regressor from Scikit-learn [20] was used to generate the trees that were used for this task.

**Table 1: Inputs and output for decision tree regressor.**

| Input (i) | Input (ii) | Input (iii) | Input (iv) | Output |
|---|---|---|---|---|
| Box Size | Maximum Space Remaining | Minimum Space Remaining | Mean Space Remaining | Bin Space Remaining |

The decision was made to use four inputs to train a decision tree regressor with the combined total remaining bin space and weight capacity of the target bin as the output. The four inputs consisted of: (i) the total size of the selected box, given as a total of box size plus box weight, (ii) the maximum bin space available in any single bin as a combined total of size and weight (but not including empty bins), (iii) the minimum bin space available in any bin (but not including full bins or infeasible bins), and (iv) the mean bin space available across all partially-filled bins.

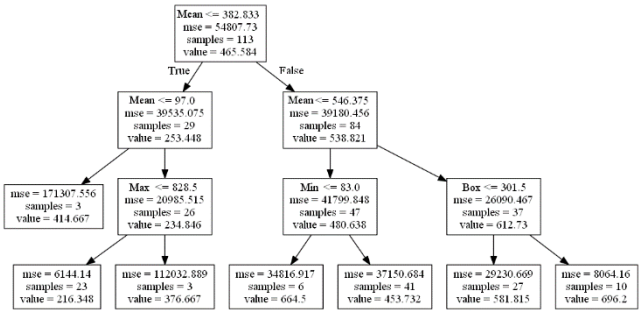


**Figure 3: Simple Tree**

Two versions of the decision tree were generated, a simplified and more generalised shallow tree limited to a maximum depth of three and a minimum leaf size of three (**Fig. 3**) and a more complex and complete tree with no restrictions (**Fig. 4**). The simplified tree was expected to be more robust when given problems of different complexities, though the complex tree might well perform better on problems that are very similar to the training problem.

Once the trees were generated, they were used in a mutation function as part of a genetic algorithm (GA), as an alternative to the standard mutation operator. This function operated by selecting a box at random and removing it from the bin it was located in. The state of the problem was then analysed for the four tree inputs and the tree queried. This returned the amount of available space to look for in a bin and found the bin that most closely matched this value. The box was then added to that bin.

The GA used was a standard Genetic Algorithm function from the Platypus library for Python [21]. This used a population size of 100 solutions coded as lists of integer strings, with simulated binary crossover (SBX) and tournament selection with a tournament size of 2. The standard mutation operator made use of the problem encoded as Gray code to perform a bit flip mutation with a probability equal to $1/n$ where $n$ is the chromosome length. This results in, on average, one member of the population being mutated at a single point each generation.

Whether the GA should use the standard mutator or the human-derived mutator (HDM) was determined by probability, with three different probabilities tested after initial trial runs. The three probabilities used were a control condition in which no human-derived mutation was used (**No HDM**), one in which 10% of human-derived mutation was used (**HDM 0.1**), and one in which 40% of human-derived mutation was used (**HDM 0.4**).

Although the games had consisted of 4-bin, 6-bin, and 8-bin problems these were too small to be a good test of the

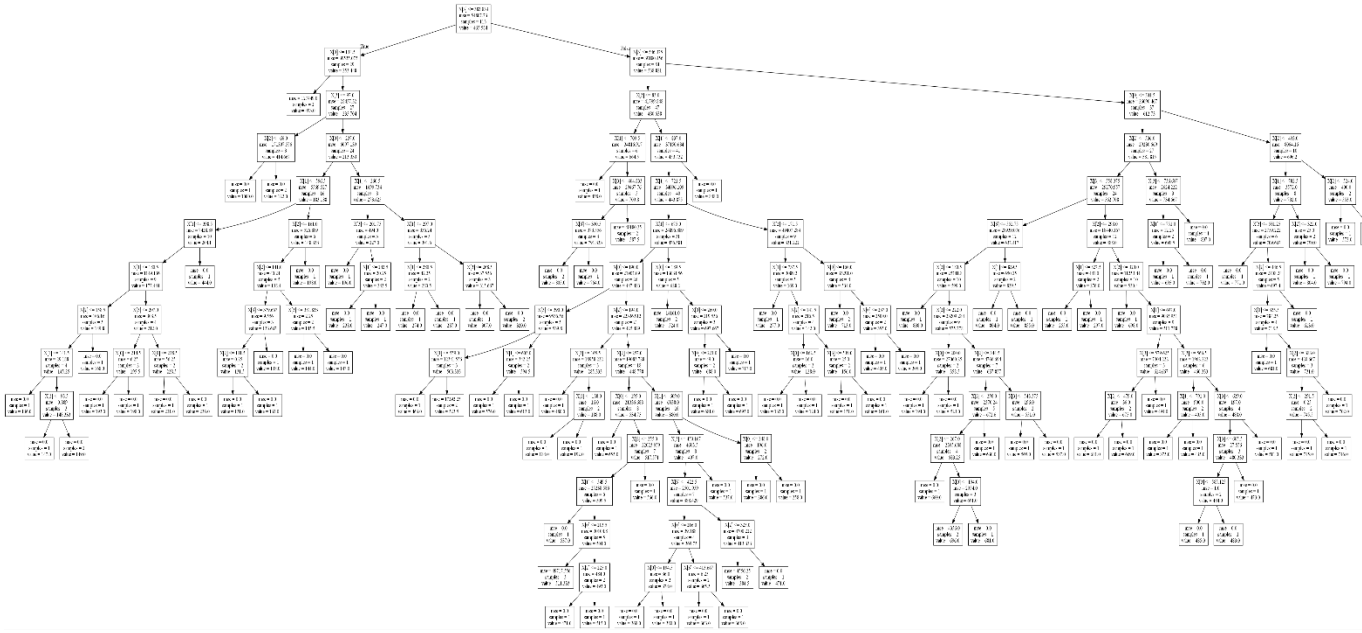

**Figure 4: Complex Tree. The tree is too detailed to be easily readable but is included to aid visualisation.**

methodology. The proposed size of the problem for the EA to solve was determined as a problem that would be unfeasible for a human to solve, but not so large as to require a supercomputer to run the genetic algorithm. After some test runs a problem size of 600 boxes with 300 bins was decided upon. This was 50 times the size of the medium 6-bin problem that only a handful of the players had completed.

In order to make a fair test, and given the stochastic nature of GA, each condition was run 30 times. After a trial run it was seen that the GA only started finding feasible solutions after about 10,000 function evaluations, and so it was decided to let the GA run for 40,000 function evaluations each run with a population of 100.

During the testing phase an additional tree was generated and tested that used only input from the poorer players, but (as expected) this achieved worse results and generated fewer feasible solutions so this was not pursued further.

In addition to recording the average score and the best scoring solution among the population, the number of feasible solutions (i.e. those that do not violate the problem constraints) were also recorded.

## 3 RESULTS AND DISCUSSION

### 3.1 Simple Tree

The first experimental results from running the GA with a mutator based on the simple tree are shown in **Figure 5** and **Figure 6**. For each run the average (mean) score across the feasible population, the best score in the population, and the percentage of feasible solutions were recorded. The 30 runs were then averaged and compared.

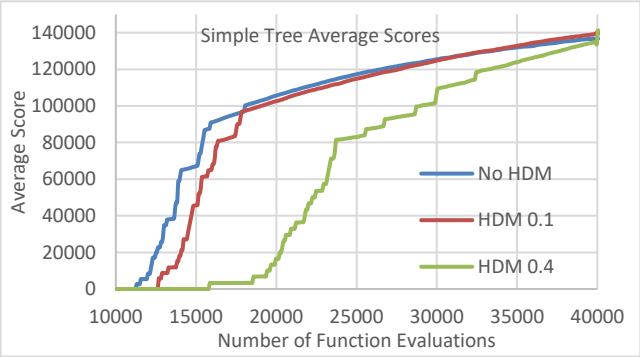


**Figure 5: Simple Tree Average Scores**

In this test it was found that the **No HDM** condition converged faster than the other two conditions both based on the average population score (**Fig. 5**) and the best population score (**Fig. 6**). However, none of the three conditions found a feasible solution until after at least 10,000 generations had passed.

While the **No HDM** convergence contrasted strongly with the **HDM 0.4** condition, it was far less noticeable when compared against the **HDM 0.1** condition. However, the **HDM 0.1** condition overtook the **No HDM** condition before the full run had finished and ended with better results in both categories.

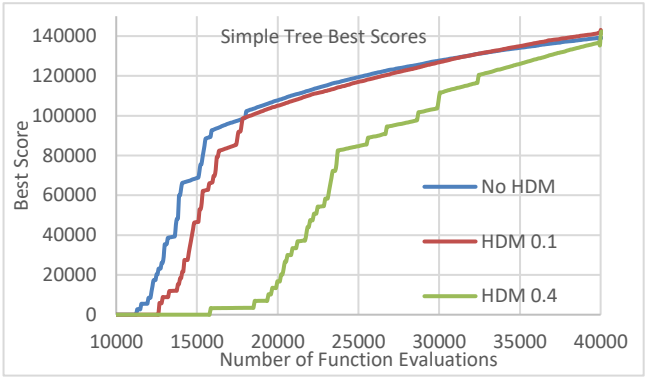


**Figure 6: Simple Tree Best Scores**

From looking at more detailed results after all runs were ended (**Table 2**) it is apparent that with regards to both mean and minimum average score and best score **HDM 0.1** was consistently better across the 30 runs than both **No HDM** and **HDM 0.4**. Although **HDM 0.4** was able to achieve both the highest maximum average score and maximum best score, it also achieved the lowest minimum scores in both categories as well showing the greatest variance.

**Table 2: Simple Tree End Results**

| | | Average Score | Best Score | % Feasible Solutions |
|---|---|---|---|---|
| **No HDM** | Mean | 137,034.25 | 139,316.5 | 45.4 |
| | Max | 142,379.49 | 145,314 | 59 |
| | Min | 130,896.78 | 133,340 | 31 |
| **HDM 0.1** | Mean | **139,364.64** | **141,553.5** | 50.9 |
| | Max | 144,952.29 | 147,540 | 61 |
| | Min | **133,456.30** | **135,430** | **36** |
| **HDM 0.4** | Mean | 135,189.16 | 136,921.9 | **69.7** |
| | Max | **148,354.47** | **149,382** | **84** |
| | Min | 113,816.17 | 116,652 | **36** |

The most interesting difference apparent in **Table 2** stems from the percentage of feasible solutions in the final population; as the amount of human-derived mutation increases the run produces a larger percentage of feasible solutions.

For each of the three factors (Average Score, Best Score, and Percentage Feasible Solutions) across the three groups (**No HDM**, **HDM 0.1**, and **HDM 0.4**) in each data set of 30 runs a single factor ANOVA was carried out, all of which found the results differed significantly (Average Score $p = .026$; Best Score $p = .009$; Percentage Feasible Solutions $p < .001$).

F-Tests were carried out to reveal any unequal variances before two-sample t-Tests were carried out. These revealed significant differences between **No HDM** and **HDM 0.1** for Average Score ($t(58) = -2.96$, $p = .004$), Best Score ($t(58) = -2.89$,

p = .005), and Percentage Feasible Solutions (t(58) = -3.29, p = .002).

When comparing **No HDM** with **HDM 0.4** significant differences were only found between the Percentage Feasible Solutions (t(51) = -10.86, p < .001), while comparing **HDM 0.1** with **HDM 0.4** found significant differences in Average Score (t(36) = 2.33, p = .025)), Best Score (t(36) = 2.66, p = .012), and Percentage Feasible Solutions (t(47) = -8.76, p < .001).

The tests showed **HDM 0.1** to have performed significantly better in all three areas over the **No HDM** standard GA after 40,000 function evaluations, while **HDM 0.4** had a significantly greater percentage of feasible solutions in the population pool.

## 3.2 Complex Tree

The complex tree results showed a similar pattern to the simple tree, though less pronounced with regards to **HDM 0.4**. As can be seen in **Figure 7** and **Figure 8** the **No HDM** condition converged faster than the other two groups both with regards to best score and average score, though only marginally faster than **HDM 0.1**.

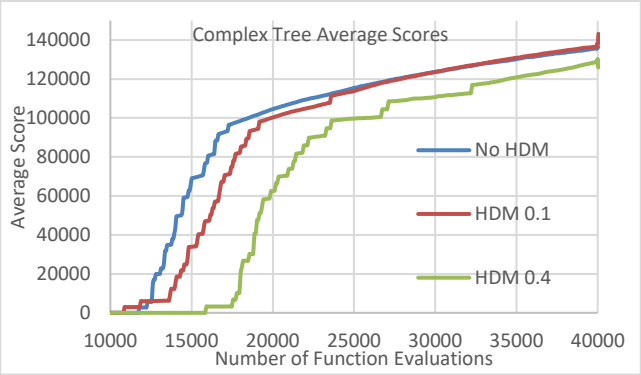


**Figure 7: Complex Tree Average Scores**

As with the simple tree, the **HDM 0.1** condition performed in a very similar way to the standard GA (**No HDM**). The average end results across the 30 runs after the 40,000 generations can be seen in **Table 3**.

**Table 3: Complex Tree End Results**

| | | Average Score | Best Score | % Feasible Solutions |
|---|---|---|---|---|
| **No HDM** | Mean | 135,841.06 | 137,928.7 | 44.1 |
| | Max | 141,996.30 | 144,170 | 56 |
| | Min | 130,134.70 | 132,116 | 32 |
| **HDM 0.1** | Mean | **136,841.64** | **138,949.5** | 50 |
| | Max | **143,212.65** | **144,986** | 60 |
| | Min | **131,380.34** | **133,540** | **35** |
| **HDM 0.4** | Mean | 128,657.76 | 131,067.7 | **66** |
| | Max | 142,820.99 | 144,534 | **80** |
| | Min | 113,161.62 | 117,424 | 26 |

Single factor ANOVA were carried out for each of the three factors (Average Score, Best Score, and Percentage Feasible Solutions) across the three groups (**No HDM**, **HDM 0.1**, and **HDM 0.4**), and again all three found significant differences in the means (Average Score p < .001; Best Score p < .001; Percentage Feasible Solutions p < .001).

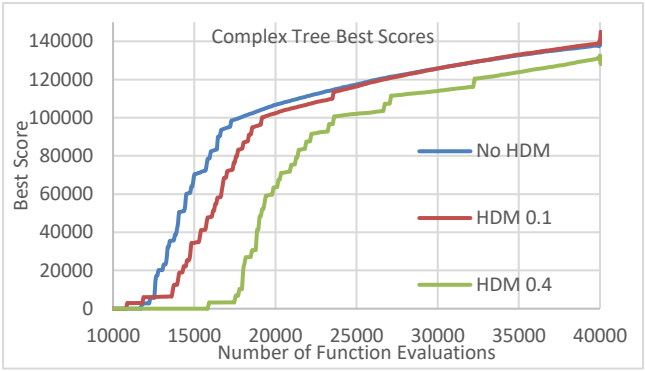


**Figure 8: Complex Tree Best Scores**

After testing for unequal variance between each pair of conditions two-sample t-Tests were run. They found no significant difference between the Average Score and Best Score of the **No HDM** and **HDM 0.1** groups (t(58) = -1.34, p = .19 and t(58) = -1.37, p = .17 respectively), though Percentage Feasible Solutions (t(58) = -3.85, p < .001) did differ significantly in favour of **HDM 0.1**.

When comparing **No HDM** against **HDM 0.4** all three factors differed significantly; Average Score (t(35) = 4.31, p < .001)) and Best Score (t(36) = 4.57, p < .001) in favour of **No HDM**, and Percentage Feasible Solutions (t(36) = -7.36, p < .001) in favour of **HDM 0.4**.

The comparison between **HDM 0.1** and **HDM 0.4** yielded similar results, with **HDM 0.1** having a significantly higher Average Score (t(36) = 4.86, p < .001) and Best Score (t(38) = 5.18, p < .001) but a significantly lower Percentage Feasible Solutions (t(39) = -5.25, p < .001).

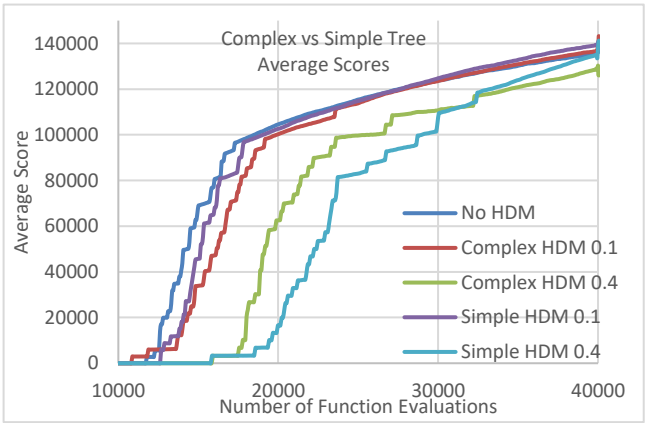


**Figure 9: Complex Tree vs Simple Tree Average Score**

### 3.3 Simple Tree vs Complex Tree

The results from the simple tree runs were then compared to the equivalent results from the complex tree. No significant difference was found between the Percentage Feasible Solutions when comparing both the **HDM 0.1** ($t(58) = 0.54$, $p = .59$) and **HDM 0.4** ($t(50) = 1.10$, $p = .27$) conditions against themselves.

When looking at the Average Score, the simple tree performed significantly better in both the **HDM 0.1** ($t(58) = 3.07$, $p = .003$) and **HDM 0.4** ($t(58) = 2.82$, $p = .007$) conditions. This was also the case with regards to the Best Score, with the simple tree **HDM 0.1** ($t(58) = 3.22$, $p = .002$) and **HDM 0.4** ($t(58) = 2.70$, $p = .009$) conditions outperforming the complex tree.

Both heuristics that derived from the simple tree therefore outperformed the heuristics derived from the more complex tree. From looking at a graphical comparison of the average solution value (**Fig. 9**) and best solution value (**Fig. 10**) over function evaluations, it can be seen that the simple tree **HDM 0.1** condition has the fastest convergence after the **No HDM** baseline condition, while the simple tree **HDM 0.4** has the slowest convergence.

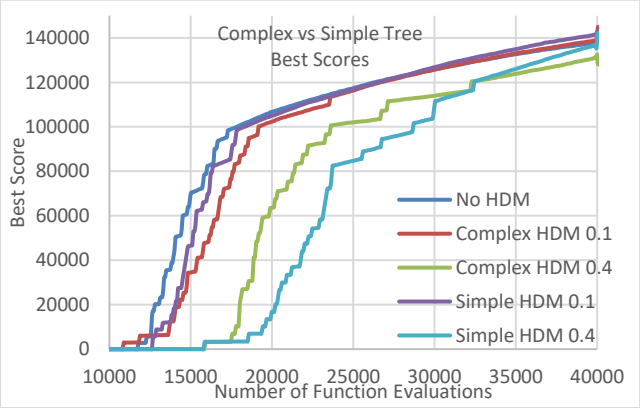


**Figure 10: Complex Tree vs Simple Tree Best Score**

### 3.4 Effect of Problem Size

Given the size of the problem and the limited number of function evaluations allotted, it was decided to see how the different techniques performed on a range of problem sizes. The initial problem size was set at 4 bins, the same size of problem that the users had played, and then doubled until reaching approximately halfway towards the 300-bin problem size tested above. The 4-bin problem with 8 boxes has an easily enumerable search space of $4^8 = 65{,}536$ possible combinations, but each doubling in size causes the problem space to grow exponentially.

To give the algorithm a greater chance to find the optimum each condition was given 200,000 function evaluations and run 50 times instead of 30. The simple tree was selected due to performance, and the comparison was expanded by the addition of a fourth condition involving the use of the human-derived mutation 100% of the time, **HDM 1.0**. As the maximum score for each problem is known, the scores were normalised to allow performance comparison between problem sizes.

On the smaller problems the algorithms that make more use of the human-derived mutation perform better with regards to average score across the population (**Fig. 11**), but the performance of the **HDM 1.0** condition soon drops off. Both the **HDM 0.1** and **HDM 0.4** conditions outperform **No HDM** across all problem sizes with regards to average score.

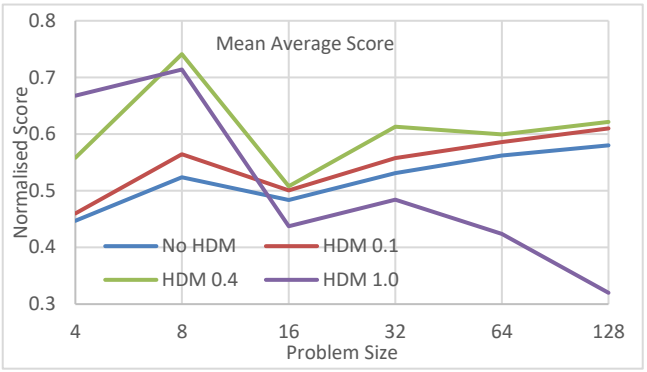


**Figure 11: Mean Average Score as problem size increases. The problem size axis uses a base 2 logarithmic scale.**

**HDM 1.0** struggles to find the best score (**Fig. 12**) across all problem sizes, while the other three conditions do very well with the smallest two problems and then abruptly drop into a gradual decline. **No HDM**, **HDM 0.1**, and **HDM 0.4** all found the 4-bin optimum every run, while **HDM 0.4** was also able to find the optimum for the 8-bin problem every run. From the 16-bin problem onwards none of the conditions were able to come close to reaching the optimum.

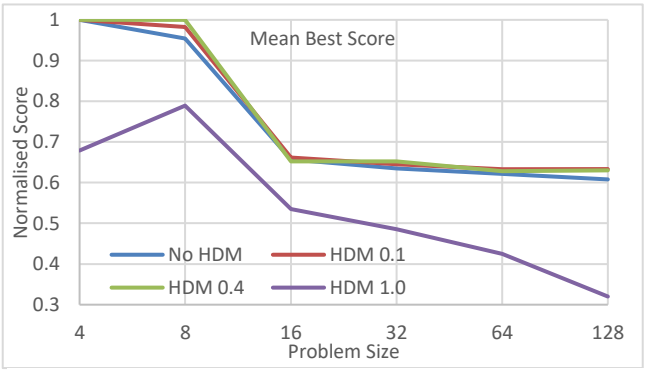


**Figure 12: Mean Best Score as problem size increases. The problem size axis uses a base 2 logarithmic scale.**

### 3.5 Discussion

The game presented the players with a simplified version of the bin-packing problem, due to the limitations of user fatigue and attention. Despite this, the gamification of this problem and the derivation of heuristics from analysis of gameplay has been shown to improve the quality of solutions discovered by an evolutionary algorithm on a much larger problem. This suggests that the learned heuristics are scalable beyond the initial problem

formulations and indicate significant promise for the method to be used on real-world scale operational research problems.

The simple tree was seen to outperform the complex tree and it is possible that this is due to overfitting of the initial problems by the complex tree, which could potentially lead to less scalability for the algorithm.

The rate of application of HDM revealed a synergy between HDM and the standard mutation that was not expected before the experiments were ran. It is clear that increased usage of HDM (i.e. 40%) results in a greater number of feasible solutions, but that the quality of the solutions overall is poorer than with the smaller 10% HDM mutation.

The users that had played the bin-packing game that the trees were generated from had been encouraged by the framing of the game to keep their solutions feasible, which could explain why using the human-derived mutation kept a larger percentage of feasible solutions in the population.

Having a larger number of feasible solutions in the population favours the production of feasible offspring, but perhaps sometimes precludes the use of infeasible solutions in the short term to generate better feasible solutions in the long term. Given the formulation of the problem a single move can turn a good solution into an infeasible one and vice versa.

As a natural function of the problems generated for the game the human-derived mutation was mainly obtained from the middle parts of the problem-solving process. There might well be different processes employed by the users near the start or end of the solution of a problem. This might mean that a better approach could be to derive algorithms for various stages of the problem and switch to them as the state of the population changes, though this could also add further complexity and make the approach less robust.

The simple tree gave a more generalised and approximate output than the complex tree, and this worked better in almost every area. This is not too surprising a result given the dangers of overfitting that could come from using the complex tree.

It is worth noting that the 300-bin problem has an achievable maximum score of 300,000 and none of the results within 40,000 function evaluations were able to reach even half of that. This is not surprising given the size of the search space ($300^{600}$) and the limited number of function evaluations used. It could be seen when looking at the effect of the problem size on performance that even relatively small problems proved too difficult to reach even 70% of the global optimum.

In the HUGS system mentioned in [17] the authors combined human input/expertise with a version of a stock-cutting/bin-packing problem. However, this was done by having the user actively guide the optimisation process by allowing them to assign priority to areas of the problem space and then running a problem-specific heuristic to optimise the problem based on that guidance. This process would then be repeated by the user until they were happy with the result.

Capturing heuristics was done through observation of user interactions rather than any systematic process and was used to improve existing problem-solving heuristics rather than generate new ones. This makes it difficult to compare the HUGS approach to the approach taken in this paper, as it would be a comparison of human involvement during the optimisation process vs. an automated approach taking advantage of human expertise. Creating a fair comparison of time and function evaluations in these circumstances would not be possible.

In future work the human-derived heuristic used in the mutation operator of the GA in this paper could be tested against the standard algorithms normally employed to solve the bin-packing problem.

## 4 CONCLUSIONS

In this paper a bin-packing game was proposed and created, designed to capture human problem-solving heuristics. Data captured from players of the game was processed to train a decision tree regressor. This tree learned which bin to put a given box into based on the general state of the problem at that time. The general problem state was defined by mean, minimum, and maximum available space across all partially-filled bins.

Two versions of this human-derived tree generated by the machine learning process were then used within a mutation operator for a GA, and used for a varying percentage of the time to make three different conditions; **No HDM** in which the human-derived mutation was not used, **HDM 0.1** in which the human-derived mutation was used 10% of the time, and **HDM 0.4** in which the human-derived mutation was used 40% of the time.

After running each condition 30 times for 40,000 function evaluations for each tree it was found that using a simple tree for 10% of the time instead of the standard mutation operator achieved significantly better results both by score and feasibility than using just the standard mutation operator or using the human-derived mutation 40% of the time.

The use of gamification and machine learning to capture human problem-solving behaviour for use within an evolutionary algorithm raises the prospect of other human-EA hybrids that are able to make use of the intuition and domain expertise of humans with the fast-global search of the evolutionary approach. The fact that behaviours discovered by non-experts on small problems were able to be translated to a large-scale problem to improve performance is particularly worthy of mention.

## REFERENCES

[1] 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, Dec. 2007.

[2] D. S. Johnson, 'Fast algorithms for bin packing', *Journal of Computer and System Sciences*, vol. 8, no. 3, pp. 272–314, Jun. 1974.

[3] J. O. Berkey and P. Y. Wang, 'Two-Dimensional Finite Bin-Packing Algorithms', *J Oper Res Soc*, vol. 38, no. 5, pp. 423–429, May 1987.

[4] H. I. Christensen, A. Khan, S. Pokutta, and P. Tetali, 'Approximation and online algorithms for multidimensional bin packing: A survey', *Computer Science Review*, vol. 24, pp. 63–79, May 2017.

[5] E. G. Coffman, M. R. Garey, and D. S. Johnson, 'Approximation Algorithms for Bin-Packing — An Updated Survey', in *Algorithm Design for Computer System Design*, Springer, Vienna, 1984, pp. 49–106.

[6] S. Martello and D. Vigo, 'Exact Solution of the Two-Dimensional Finite Bin Packing Problem', *Management Science*, vol. 44, no. 3, pp. 388–399, Mar. 1998.

[7] E. Falkenauer, 'A hybrid grouping genetic algorithm for bin packing', *J Heuristics*, vol. 2, no. 1, pp. 5–30, Jun. 1996

[8] J. F. Gonçalves and M. G. C. Resende, 'A biased random key genetic algorithm for 2D and 3D bin packing problems', *International Journal of Production Economics*, vol. 145, no. 2, pp. 500–510, Oct. 2013.
[9] M. A. Kaaouache and S. Bouamama, 'Solving bin Packing Problem with a Hybrid Genetic Algorithm for VM Placement in Cloud', *Procedia Computer Science*, vol. 60, pp. 1061–1069, Jan. 2015.
[10] E. K. Burke, M. R. Hyde, and G. Kendall, 'Evolving Bin Packing Heuristics with Genetic Programming', in *Parallel Problem Solving from Nature - PPSN IX*, Springer, Berlin, Heidelberg, 2006, pp. 860–869.
[11] X. Li, K. Deb, and Y. Fang, 'A derived heuristics based multi-objective optimization procedure for micro-grid scheduling', *Engineering Optimization*, vol. 49, no. 6, pp. 1078–1096, Jun. 2017.
[12] P. Ryser-Welch, J. F. Miller, and S. Asta, 'Generating Human-readable Algorithms for the Travelling Salesman Problem Using Hyper-Heuristics', in *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, New York, NY, USA, 2015, pp. 1067–1074.
[13] E. López-Camacho, H. Terashima-Marin, P. Ross, and G. Ochoa, 'A unified hyper-heuristic framework for solving bin packing problems', *Expert Systems with Applications*, vol. 41, no. 15, pp. 6876–6889, Nov. 2014.
[14] K. Sim, E. Hart, and B. Paechter, 'A Lifelong Learning Hyper-heuristic Method for Bin Packing', *Evolutionary Computation*, vol. 23, no. 1, pp. 37–67, Feb. 2014.
[15] C. Tijus *et al.*, 'Human Heuristics for a Team of Mobile Robots', in *2007 IEEE International Conference on Research, Innovation and Vision for the Future*, 2007, pp. 122–129.
[16] G. Kefalidou, G. Kefalidou, and T. C. Ormerod, 'The Fast and the Not-So-Frugal: Human Heuristics for Optimization Problem Solving', p. 7.
[17] G. W. Klau, N. Lesh, J. Marks, and M. Mitzenmacher, 'Human-guided search', *J Heuristics*, vol. 16, no. 3, pp. 289–310, Jun. 2010.
[18] C. Murawski and P. Bossaerts, 'How Humans Solve Complex Problems: The Case of the Knapsack Problem', *Scientific Reports*, vol. 6, p. 34851, Oct. 2016.
[19] B. M. Good, S. Loguercio, O. L. Griffith, M. Nanis, C. Wu, and A. I. Su, 'The Cure: Design and Evaluation of a Crowdsourcing Game for Gene Selection for Breast Cancer Survival Prediction', *JMIR Serious Games*, vol. 2, no. 2, Jul. 2014.
[20] F. Pedregosa *et al.*, 'Scikit-learn: Machine Learning in Python', *Journal of Machine Learning Research*, vol. 12, p. 2825−2830, Oct. 2011.
[21] 'Platypus - Multiobjective Optimization in Python — Platypus documentation'. [Online]. Available: https://platypus.readthedocs.io/en/latest/#. [Accessed: 06-Feb-2019].