3D Packing Problem Using Evolutionary Computations

Giovanni Scialla

***Abstract*—In this work, it's going to be addressed the Three-Dimensional Packing Problem (3DPP), an optimization challenge with diverse applications spanning logistics, manufacturing, and resource allocation. This paper presents a comprehensive investigation into the efficacy of Evolutionary computations using Genetic Algorithms (GA) in comparison to a more classical greed-search based approach using Mixed-Integer Programming (MIP), into addressing the complexities inherent to the 3D packing problem.**

I. Introduction

THE primary objective of this research is to evaluate and compare the performance of two different programming paradigms, Mixed-Integer Programming (MIP) and Genetic Algorithms (GA) in optimizing packing configurations, considering real-world constraints and practical applications. MIP, a well-established mathematical programming technique, formulates the 3DPP as a mathematical model with integer decision variables, providing exact solutions within reasonable computational time. On the other hand, GA, a nature-inspired heuristic algorithm, leverages evolutionary principles to evolve potential solutions, demonstrating adaptability to complex problem spaces. The paper discusses the trade-offs between solution quality and computational efficiency, providing insights into the practical applicability of each approach for different problem contexts. This research contributes to the existing body of knowledge on optimization techniques for the 3D packing problem, offering valuable insights for practitioners seeking the most suitable approach based on problem characteristics. The findings aim to inform decision-makers in industries relying on efficient packing strategies, ultimately facilitating enhanced resource utilization and operational efficiency. Before diving directly into the 3D world, a much simpler 2D packing case has been also explored. For the sake of simplicity, in this work it's only covered the 3D case but still, all the implementations explained below can be adapted to the 2D case by simply removing the Depth (D) dimension and considering the rotation only along one axis instead of three.

II. Problem Statement

In the vanilla version of the 3D Bin Packing Problem, a set of n items of different sizes must be packed into a finite number of bins or containers, each of a fixed given capacity, in a way that minimizes the number of bins. In our case we are going to address a variant of this problem where we have only one bin (the Container), and a number n of items. The goal here is to understand whether it is possible to contain all the items inside the bin, and in that case, the algorithm should return a possible arrangement of them.

A. Objective Function

The objective function is defined as a maximization function since the goal is to maximize the number of objects inside the container,thus it is defined as:

$$max \sum_{i=1}^{n} \sum_{k=1}^{m_i} s_{ik} \tag{1}$$

Where the binary variable s_{ik} specify whether item i is taken with rotation k. n is the number of items and m_i the possible rotations of each item.

B. MIP Constraints

When dealing with the Packing Problem using a MIP approach, a set of binary variables $l_{ij}, r_{ij}, u_{ij}, o_{ij}, b_{ij}, f_{ij}$ is needed to specify the relative position of object i regarding object j. Since there can be only one mutual position between two objects, the following constraint must be added.

$$l_{ij} + r_{ij} + u_{ij} + o_{ij} + b_{ij} + f_{ij} = 1 \tag{2}$$

Also, in this case both the item i and j must be part of the solution.

$$\sum_{k=1}^{m_i} s_{ik} + \sum_{k=1}^{m_j} s_{jk} = 1 + 1 \tag{3}$$

For a detailed list of all the derived constraints that must be defined in order to maintain the validity of a solution can be found in the Appendix A at the end of the elaborate.
With an evolutionary approach instead, there is no need to define those constraints because genetic algorithms can naturally handle them while maximizing the fitness function. The flexibility of evolutionary computations make them able to achieve a quality solution even in this complex, non-linear environment.

III. Genetic Algorithms (GA) for Solving the 3D Packing Problem

A. Representation of Solution

The first thing to define is a way to represent an individual of our genome (that will be a possible solution of the problem). The representation is completely arbitrary and its definition will have an impact in the implementation of the genetic operators. In this work I present two possible representations that depends on whether we want to allow each item to be rotated or not. In the simpler case, we do not allow rotation so the solution can be simply represented as:

$$genome = [(x_1, y_1, z_1), ..., (x_n, y_n, z_n)] \tag{4}$$

where each set of three coordinates represent the position of the bottom-left corner of the item inside the Container coordinate space.
If we also want to account for rotations, we have to add 3 binary variables inside each gene that tells whether the item i is rotated in respect to axes x, y or z

$$genome = [(x_1, y_1, z_1, r_{1x}, r_{1y}, r_{1z}), ...] \tag{5}$$

Once we have a way to represent a possible individual, which will be nothing less than a possible configuration of our items inside the box. The next step is to instantiate a population of different random individuals in order to begin the evolutionary process.

B. Fitness Function

The second ingredient that we need to define is the fitness function. The value of the fitness score assigned at each individual will guide the evolutionary process towards better solutions of our problem. In this work I have used a fast vectorization approach to compute the maximum non-overlapped area occupied by the items inside the Box. A penalty function can be added whenever an item is overlapped with another one. Solutions that present items well distantiated from each others would get an higher fitness score, thus they are encouraged to survive as the process goes on. Details of the implementation are explained in the pseudo-code 1

Algorithm 1 Fitness Function (Maximization Problem)

```
function FITNESS(genome, items, penalty)
    space ← zeros(W, H, D)
    for gene, item in zip(genome, items) do
        space[gene[0] + w, gene[1] + h, gene[2] + d ← 1
    end for
    non_overlap_area ← where(space == 1)
    fitness_score ← space[non_overlap_area]
    if penalty then
        overlap_area ← where(space > 1)
        penalty_score ← space[overlap_area]
        return (fitness_score − penalty_score)
    end if
    return fitness_score
end function
```

C. Selection

The selection operator is the first genetic operator that we have to define. There are various methodologies available from the literature (random selection, rank-based selection, tournament selection, hall-of-fame selection, etc...) In this work i have used the simple roulette-wheel selection that promotes the possibility of being chosen as parents towards the most fitting individuals. This way of selecting the parents poses an high selection pressure inside the evolution. In the pseudocode 2 above, the weights obtained using the fitness score are used to choose the number of parents n from the population. During the selection process a number of best individuals (Elites) is kept inside the new population to keep the best solutions found so far alive.

Algorithm 2 Selection Operator

```
function SELECTION(population, n_parents)
    weights = Fitness(population)
    return choices(population, weights, n_parents)
end function
```

D. Crossover

A single-point crossover operator is used to recombine different genes in the new population. The recombination adds some exploitation capabilities in the process of finding the optimal solution by leveraging different dispositions of items inside the box already encoded in the actual population. In the pseudo-code 3 we can see how the point where recombine the genetic material from both the parents is picked randomly.

Algorithm 3 Crossover Operator

```
function CROSSOVER(gene_a, gene_b)
    assert len(a) == len(b)
    length ← len(gene_a)
    if length < 2 then
        return gene_a, gene_b
    end if
    p ← randint(1, length − 1)
    child_a ← gene_a[0 : p] + gene_b[p :]
    child_b ← gene_b[0 : p] + gene_a[p :]
    return child_a, child_b
end function
```

E. Mutation

Last but not least, the mutation operator adds explorative capabilities to the evolutionary process. In this case, with a certain probability, we can randomize the position and rotation of a number of items inside our solution, hopefully obtaining a more optimal displacement. The implementation details are defined in the pseudocode 4 below. A number of random genes are chosen from the individual to which the mutation will be performed. How many of these mutations will actually succeed is given by the mutation probability, a parameter that must be decided given some heuristics of the problem in hand.

IV. Experiments and Results

In this chapter, we present the results obtained from our experiments comparing Mixed-Integer Programming (MIP) and Genetic Algorithms (GA) for solving the 3D packing problem. The objective is to evaluate the performance of these two optimization approaches across three different scenarios:

1) Small Scenario: Box [3x3x3] ; items [12]
2) Medium Scenario: Box [10x10x10] ; items [20]
3) Big Scenario: Box [15x30x15] ; items [30]

These scenarios are designed to test the algorithms' capabilities. Each item is characterized by its dimensions, and the goal is to find an optimal or near-optimal packing arrangement within a 3D space. For each run of genetic algorithm, the experiments have been conducted with consistent parameters, such as population size, mutation rate, and termination criteria.

Algorithm 4 Mutation Operator

function MUTATION(genome, items, n_mutations, prob)
  for n_mutations **do**
    index $\leftarrow Random$(index)
    if $Random() \leq$ prob **then**
      rx, ry, rz $\leftarrow Random(0, 1)$
      w, h, d $\leftarrow$ Rotate(items[index])
      $x \leftarrow$ randint$(0, W - w)$
      $y \leftarrow$ randint$(0, H - h)$
      $z \leftarrow$ randint$(0, D - d)$
      genome[index] $\leftarrow (x, y, z,$ rx, ry, rz$)$
    end if
  end for
  return genome
end function

Multiple Runs	Execution Time (MIP)	Execution Time (GA)
Small Scenario	12.96s	13.24s
Medium Scenario	57.07s	16.54s
Big Scenario	223.54s	64.29s

TABLE I
HERE ARE REPORTED THE AVERAGE EXECUTION TIME TO OBTAIN A VALID SOLUTION IN ALL THE CASE SCENARIOS. THE EXECUTION TIME IS CALCULATED AS AN AVERAGE OVER MULTIPLE RUNS SINCE THE GENETIC ALGORITHM HAS A LOT OF RANDOMNESS DURING THE PROCESS.

The experiments were run on the same device and same conditions to ensure fair comparisons. Expecially for the Evolutionary approach, due to its non-deterministic component, a number of multiple runs has been conducted in order to obtain an average execution time that can be representative. Those results are shown in the table I. Here we can see how the Genetic algorithms can be a win condition, from a computational point of view, over a greed-search approach, expecially when the search space grows too large.

Some other experiments has been conducted in order to try various parameters. it's important to keep in mind that the results shown in the table II can vary abruptly, depending on the problem instances. In This particular case i made sure to take the same items for each experiments and also to carefully choose the overall items set to be small enough to fit inside the box but big enough for the problem to still be challenging from an optimization point of view.
In conclusion , for the Big Scenario, a representation of the fitness during the evolution is also provided in figure 1, along with the 3D Optimal configuration (Figure 2) found during the process. It's important to notice how the fitness grows very fast at the beginning but it converges very slowly at the end. This is due to the nature of the problem in hand. The algorithm at the start of the process it has a lot of better configuration to explore, so the fitness function will increase rapidly until convergence to a sub-optimal solution. That solution is very hard to surpass because, at the very end, only little movements of particular items are required to improve the overall score. The fitness behavior leaves space for possible improvements of the evolutionary process itself, like for example, restarting with a new population once the algorithm stagnates (but keeping some elites to not lose the best configurations found so far). Another improvement could be to use an adaptive mutation rate that can leverage between a good amount of explorative capabilities at the beginning and exploitative ones at the very end.

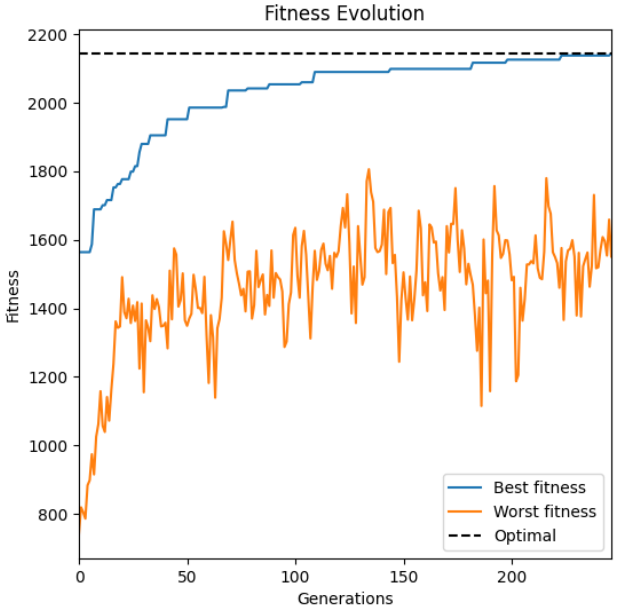

Fig. 1. The progression of the best and the worst fitness score inside the population. The best fitness will never drop due to the Elitism mechanism.

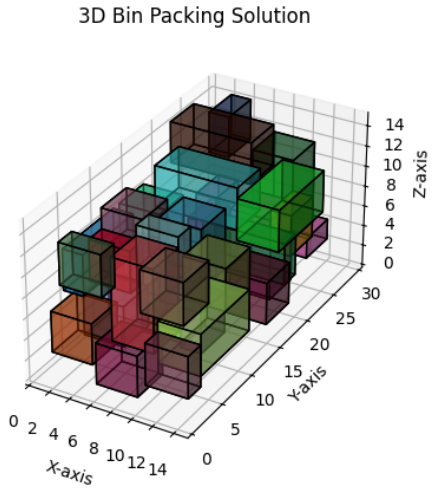

Fig. 2. A possible disposition of the items of one solution to the 3D Packing problem.

V. CONCLUSION

In conclusion, here are some potential advantages of using an evolutionary algorithm over a mixed-integer approach in the context of the 3D packing problem:

	Population	Mutations	Mutation Probability	Elites	Max. Generations	Generations	Optimal
Small Scenario	10	1	0.4	2	1000	/	No
	20	1	0.4	2	1000	/	No
	20	2	0.6	4	1000	56	Yes
	50	3	0.4	2	1000	87	Yes
	50	3	0.6	4	1000	/	No
Medium Scenario	10	1	0.4	2	1000	677	Yes
	20	1	0.4	2	1000	112	Yes
	20	2	0.6	4	1000	89	Yes
	50	3	0.4	2	1000	/	No
	50	3	0.6	4	1000	334	Yes
Big Scenario	10	1	0.4	2	1000	/	No
	20	1	0.4	2	1000	636	Yes
	20	2	0.6	4	1000	246	Yes
	50	3	0.4	2	1000	/	No
	50	3	0.6	4	1000	439	Yes

TABLE II
HERE ARE SOME EXPERIMENTS WITH VARIOUS GENETIC PARAMETERS. THE RESULTS SHOW WHETHER THE EVOLUTIONARY PROCESS WAS ABLE TO FIND AN OPTIMAL SOLUTION IN A RANGE OF 1000 GENERATIONS.

- **Flexibility** Evolutionary algorithms can naturally handle mixed-integer variables and constraints without the need for additional techniques. This makes them suitable for problems where both continuous (item position) and discrete variables (item rotations) are present.
- **Adaptability** Evolutionary algorithms can naturally handle different types of constraints,including complex and non-linear ones, making them suitable for a wide range of optimization problems, including 3D packing.
- **Exploration of Solution Space** Evolutionary algorithms are often designed to perform global optimization, searching for the best possible solution. The exploration across diverse set of solutions in the fitness landscape can be advantageous in the 3D packing problem, where finding a globally optimal solution is crucial for efficient space utilization.

While evolutionary algorithms offer certain advantages, they also have some disadvantages compared to mixed-integer approaches for solving the 3D packing problem. Here are some potential drawbacks:

- **Convergence Speed** Evolutionary algorithms, with certain problem settings, may take longer to converge to an optimal solution. This slow convergence can be a disadvantage when faster optimization methods are desired, especially time-sensitive applications.
- **Parameter Tuning** Evolutionary algorithms often involve tuning various parameters, such as population size, mutation rates, and selection modalities. The performance of the algorithm can be sensitive to the choice of these parameters, and finding the optimal set may require additional effort and experimentation.
- **Lack of Problem-Specific Knowledge** Evolutionary algorithms might not take advantage of specific problem structures or domain knowledge in the 3D packing problem. In contrast, greed-search based techniques can exploit problem-specific heuristics to enhance optimization.

In summary, the choice between an evolutionary algorithm and a mixed-integer approach depends on the specific settings of the 3D packing problem and the priorities of the optimization task. It is often beneficial to carefully analyze the problem and experiment with different algorithms to determine which approach performs better in a given context.

APPENDIX A
3D PACKING CONSTRAINTS

Listed below are all the other contrainst needed in the 3D packing problem in order to assure the integrity of the final solution.

$$l_{ij} + r_{ij} + u_{ij} + o_{ij} + b_{ij} + f_{ij} \geq \sum_{k=1}^{m_i} s_{ik} + \sum_{k=1}^{m_j} s_{jk} - 1 \quad (6)$$

$$l_{ij} + r_{ij} + u_{ij} + o_{ij} + b_{ij} + f_{ij} \leq \sum_{k=1}^{m_i} s_{ik} \quad (7)$$

$$l_{ij} + r_{ij} + u_{ij} + o_{ij} + b_{ij} + f_{ij} \leq \sum_{k=1}^{m_j} s_{jk} \quad (8)$$

These other constraints are needed to avoid placing items in places where there is no space left for them.

$$0 \leq x_i \leq W - \sum_{k=1}^{m_i} s_{ik} w_{ik} \quad (9)$$

$$0 \leq y_i \leq H - \sum_{k=1}^{m_i} s_{ik} h_{ik} \quad (10)$$

$$0 \leq z_i \leq D - \sum_{k=1}^{m_i} s_{ik} d_{ik} \quad (11)$$