Strategies for solving the Perfect Square Placement Problem

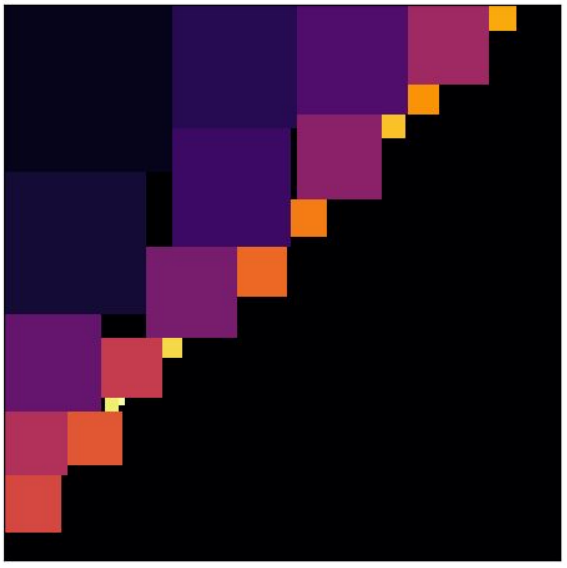
I. Greedy approach

To get an upper limit for the performance obtainable for this problem we implemented a greedy approach for solving it. Our method uses a grid as the space of all possible values from solution (placements) of a square. The algorithm iterates this space for every square that needs to be placed and chooses as the solution the first placement which satisfies the constraint that squares should not overlap each other. This approach yields a solution containing the placement coordinates for all squares and the size of a bigger square, we will call it *master square*, containing all placed squares.

The objective function of our LP problem was to minimize the master square size, meaning that the placed squares will be as tight as possible. To achieve better solutions, the implementation of the greedy choice was improved by optimizing the traversal of the grid. The traversal is done along the second diagonal in order to minimize the size of the master square.

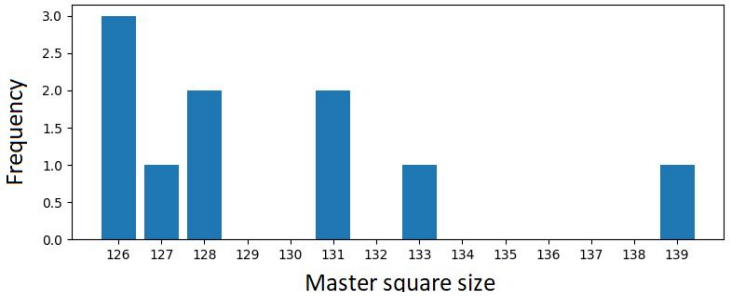
Furthermore, the optimization of the objective function is done by running the greedy algorithm multiple times:

- The first run establishes a starting point for the grid size. The solution for this iteration will consist of all squares being placed in the upper side of the secondary diagonal of the master square.



- Using the size given by the previous iteration as a baseline for the master square size, we run the greedy algorithm on decreasingly smaller sizes and attempt to fit the squares into it. For each iteration is used a decrease by one master square size, until it reaches a size in which the greedy algorithm fails to place all squares. At this point, the iterations stop and the smallest master size in which all squares are placed is considered to be the optimal solution of the objective function.

An observation worth mentioning is that the optimal value depends on the order of the squares. We ran an experiment to illustrate that. We ran 10 times the greedy algorithm that we described above and shuffled the order of the squares before each run.

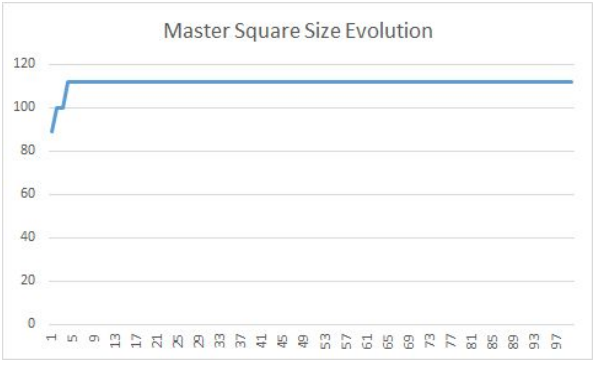


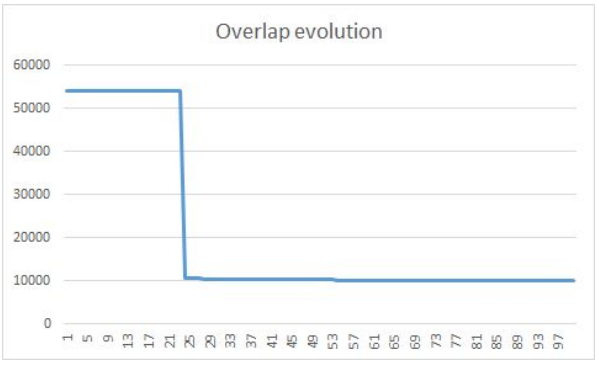
The objective function value varies a lot for the same input.

To conclude this experiment, the greedy approach finds a solution for the problem that is not optimal in most cases and is sensitive to the order of the squares.

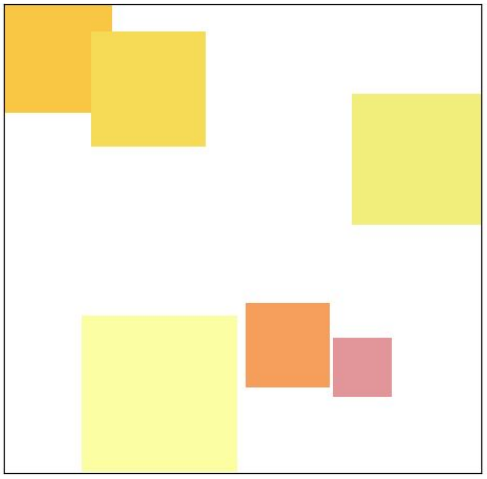
II. Genetic algorithm approach

As presented before, the search space for the perfect square placement is implicitly high-dimensional and a genetic algorithm is suitable for finding a good solution in this space due to the implicit parallelism it provides.
The design of the genetic algorithm starts with the individual representation. For the first approach regarding the genetic algorithm, the individual was an array of tuples (x, y) representing a valid configuration for the placement of the squares, meaning that we could place the squares according to an array of tuples inside a master square. The problem here was to find the right size for this master square. We used the best value obtained in the greedy approach.
Next, we chose a bi-objective fitness function. The genetic algorithm makes the selection based on two criteria: the overlap space among squares and the maximum square size the individual needs to place all the squares.





The mutation operator was implemented as changing, at random point a set of coordinates with random values. The crossover operator was to interchange two arrays of coordinates at a random cut point.
Initialization was done with arrays that place every square at the origin of the axis, having the first component of the fitness function the worst value possible and the other component, the other way around.
The selection method was the roulette and we started with a population of 100 individuals, each iteration (100 in total), “playing” the roulette another 100 times. The mutation rate was 0.1 and the crossover rate 0.6.



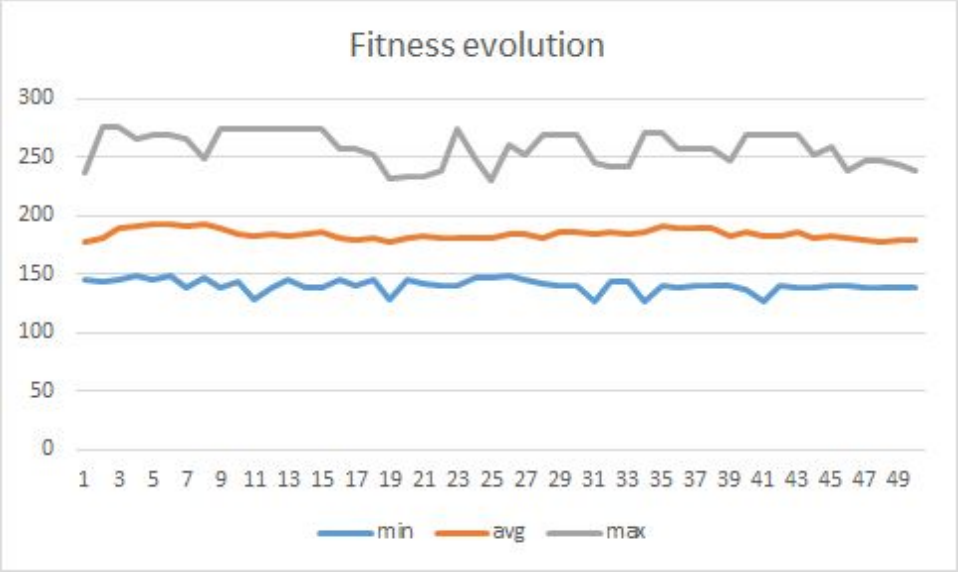
At the start of evolution: master square size 108, 37335 overlap

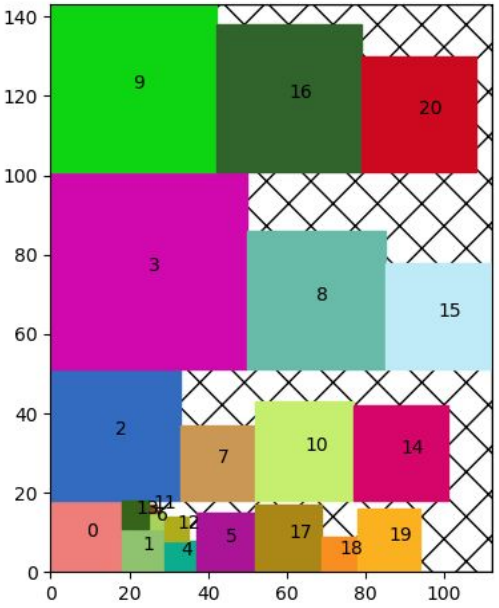
End of evolution, master square size 112, 10129 overlap

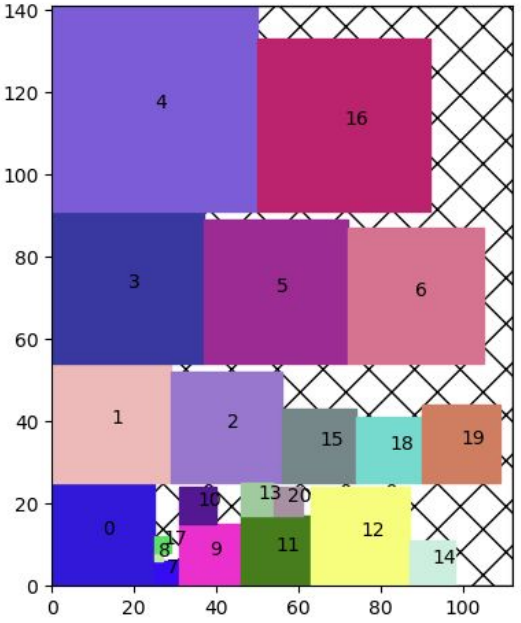
The second approach using a genetic algorithm was mostly inspired by A new metaheuristic genetic-based placement algorithm for 2D strip packing. The idea is to use the genetic algorithm for just one objective, in our case minimize the size of the master square and use a deterministic, close to the optimal algorithm for square placing. We were inspired by the results we discovered for the greedy approach: the order in which a deterministic approach gets the squares influences a lot the final solution.

Basically, we reduced our problem to a strip packing problem (using as few strips of known width to pack a set of squares) which can be solved using already implemented heuristics like priority heuristic.

In this case, the individual will be a representation of a permutation of square sizes. The fitness function is the height (number of stripes * stripe width) obtained by using the permutation of squares as input to a deterministic algorithm. The mutation and crossover operators are the classic ones for permutation representations. We experimented with ordered and partially-mapped crossover. For the mutation, we tried shuffle indexes mutation.


Fitness evolution
300
250
200
150
100
50
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49
min
avg
max


140
120
100
80
60
40
20
0
0 20 40 60 80 100
9
16
20
3
8
15
2
7
10
14
0
1
4
5
17
18
19


140
120
100
80
60
40
20
0
0 20 40 60 80 100
4
16
3
5
6
1
2
15
18
19
0
9
11
12
14