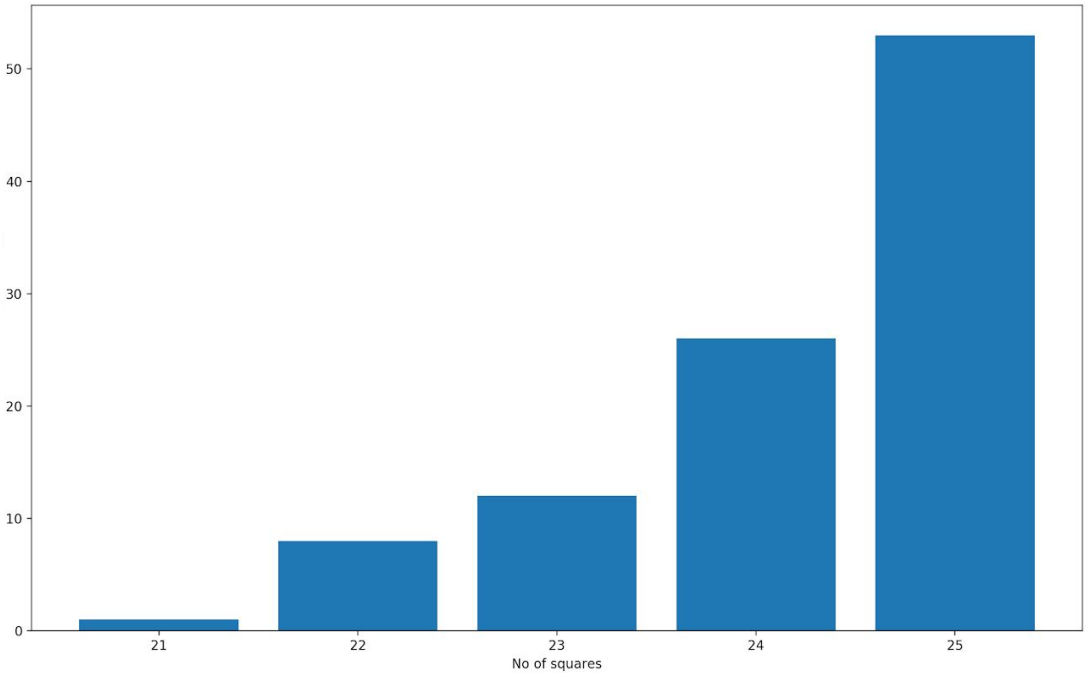
Perfect Square Placement Problem
- Advanced issues -

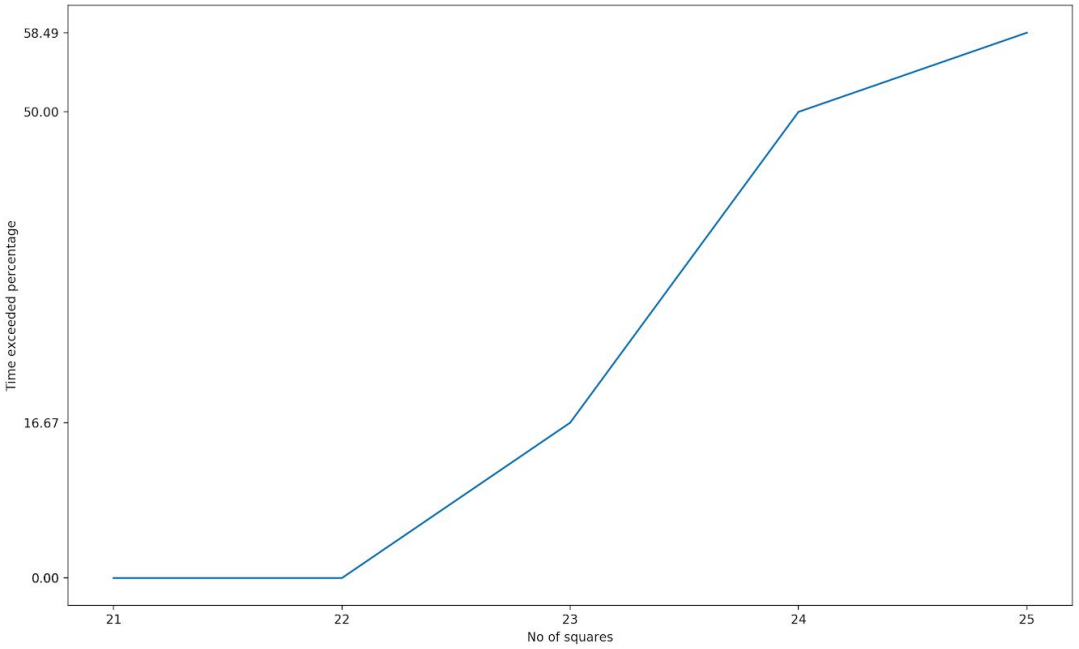
I. Runs on larger instances

We tested our optimized greedy approach on the first 100 instances from http://csplib.org/Problems/prob009/data/helmut.pdf. The number of squares to place ranges from 21 to 25.
In order to be able to run the greedy algorithm on larger instances we needed to do one more optimization. After performing first phase greedy we obtain an upper bound. Our first optimization started from a lower bound (optimal value) and tried to find a feasible solution. For larger instances this approach proved to be too time consuming. As a solution to this, we considered a binary search between the lower and upper bound and thus we obtained a logarithmic search complexity.
The first 100 instances distribution based on number of squares is the following:

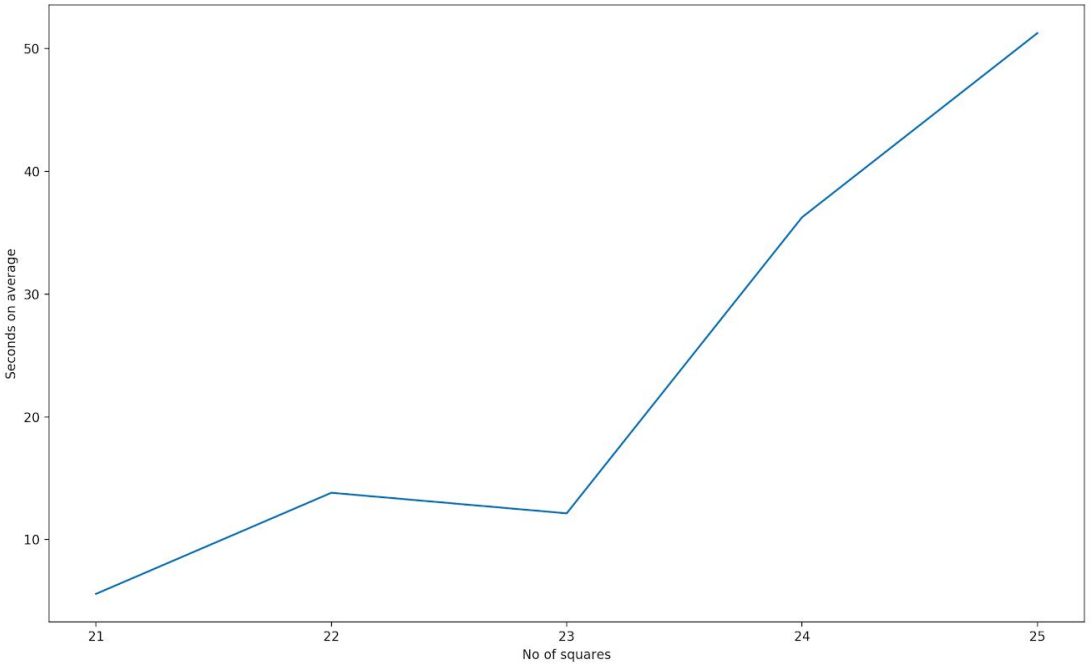


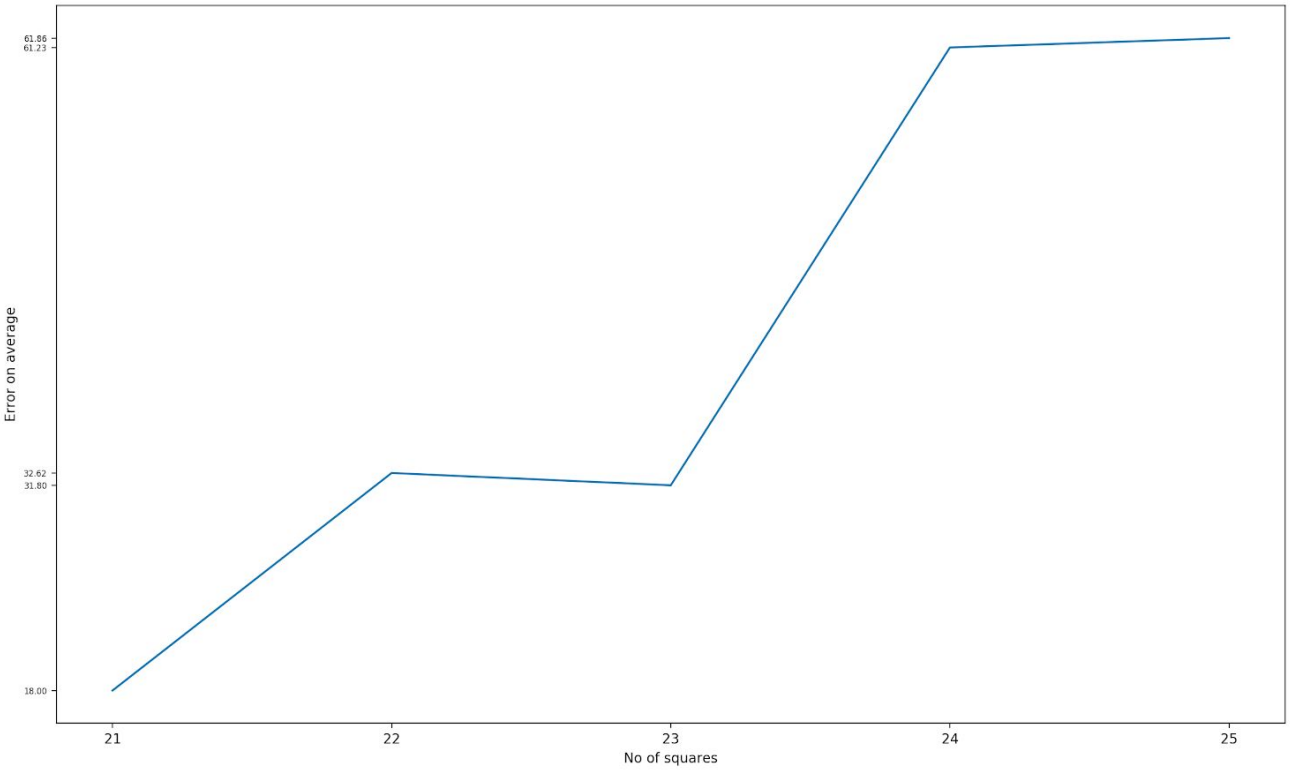
As expected for some of these instances we couldn't wait for the whole greedy approach to finish as it proved to be too expensive from the time perspective. We limited the running time of our approach to 100 seconds per instance thus the whole

experiment fitted in a 2 hour time frame. Below there is a figure which shows how many instances we stopped the algorithm from running after our proposed timeout.



We registered the time taken to find a greedy solution and how far is this solution from the optimal one in average over all instances with the same number of squares to be placed.





As expected, our algorithm error rate and time increases linearly with the number of squares to be processed.

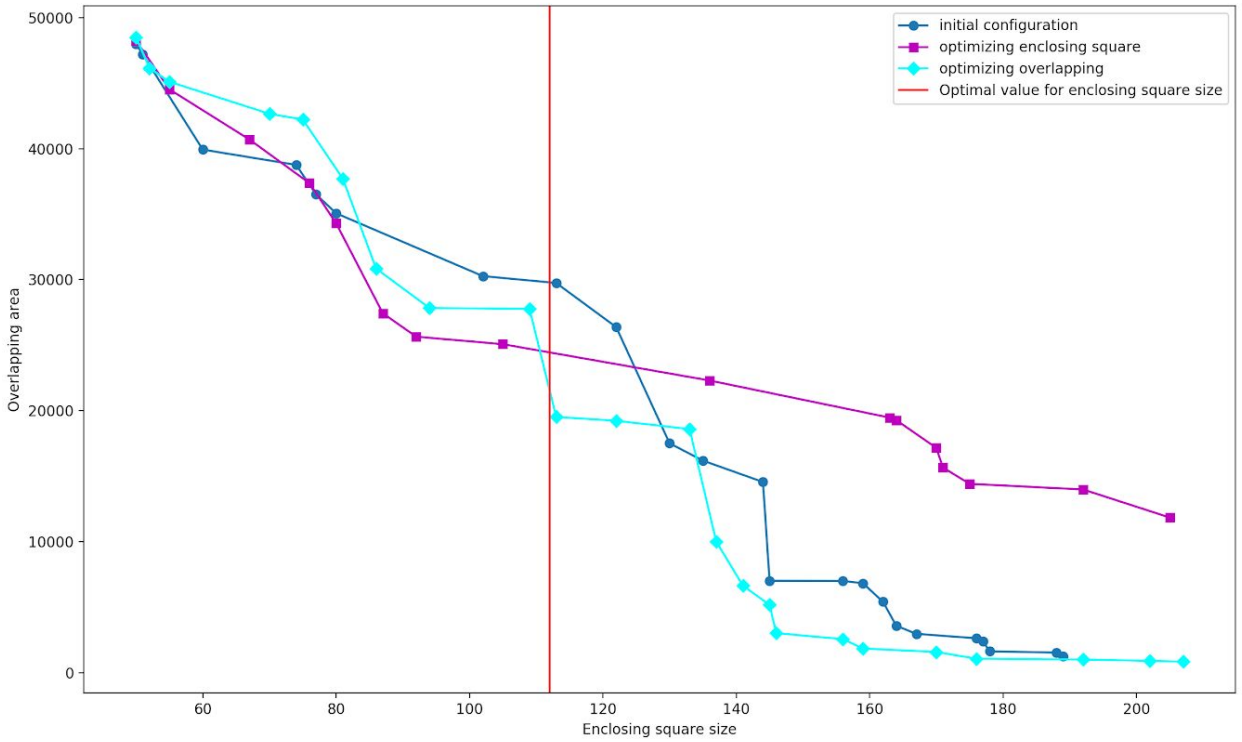
II. Automatic algorithm configuration

For this task, we decided to use the first genetic algorithm approach. For the hyperameters we chose the population size, the crossover rate, the mutation rate and the number of generations. We used for finding the best configuration the Sequential Model-based Algorithm Configuration implementation in Python available on GitHub.

Our genetic algorithm optimizes a multi-objective function, trying to minimize both the minimum enclosing square and also the area of overlaps, and as SMAC doesn't have an implementation for multi-objective optimization as well, we decided to use the algorithm to optimize configurations for each of our objective function components.

Being short on CPU power and time we allowed the SMAC optimizer to call the genetic algorithm with different configurations just 30 times. As for the objective for SMAC we sent the mean of the metric (minimum enclosing square or overlapping

area) over the hall of fame (best individuals) over the entire populations in a genetic algorithm run. To compare with our initial guess for the parameters we plotted the Pareto front for each of the configurations found by SMAC and our initial setup.

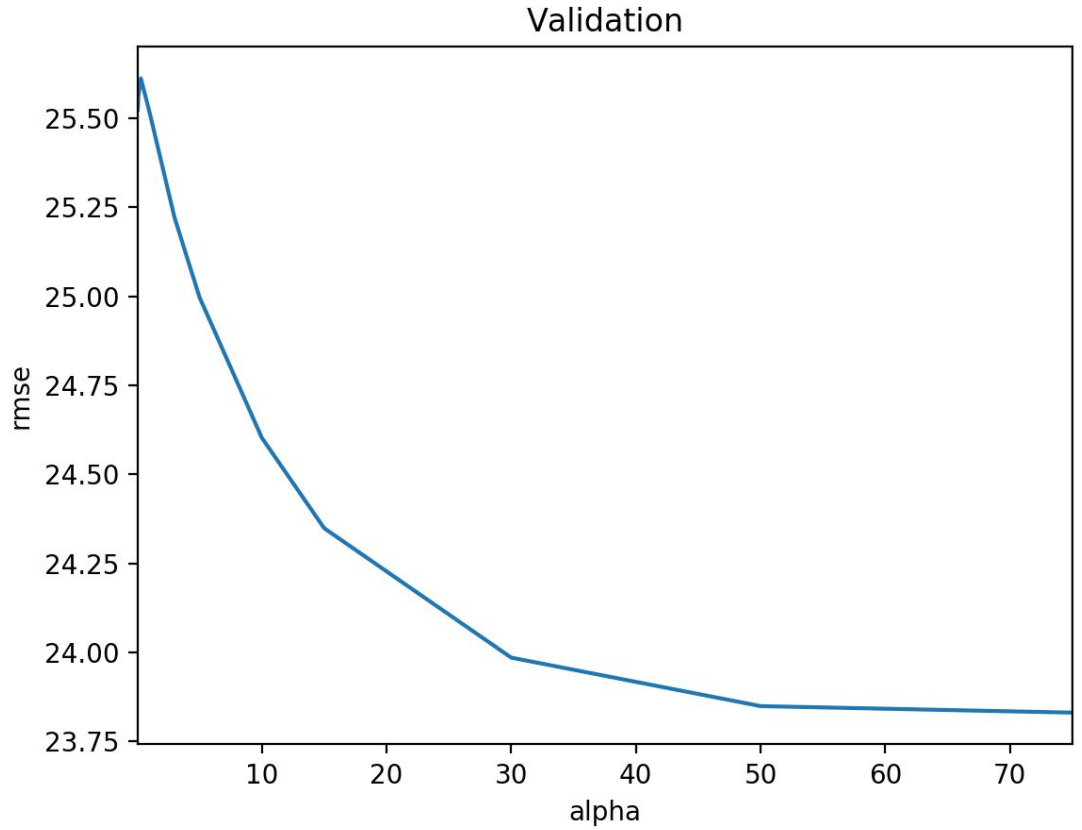


Our initial algorithm configuration was with the population size 100, the crossover rate 0.6, the mutation rate 0.2 and the number of generations 100. The first SMAC run (optimizing enclosing square) found the population size 82, the crossover rate 0.8825, the mutation rate 0.046 and the number of generations 76. The second run (optimizing overlapping) found the population size 117, the crossover rate 0.7362, the mutation rate 0.191 and the number of generations 117.

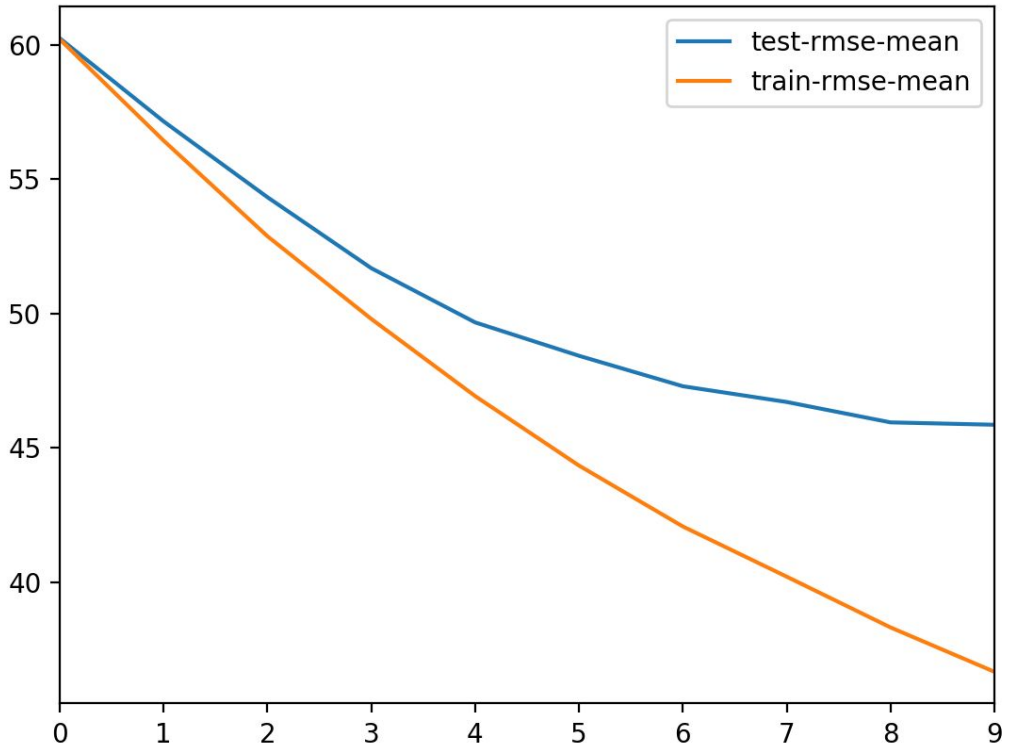
As could be seen from the figure above, optimizing the enclosing square size gives solutions that for lower than the optimal value for enclosing square have the lowest overlapping area (from the 3 sets of configurations). On the other hand, optimizing the overlapping gives solutions that for greater than the optimal value for the enclosing square have the lowest overlapping area (from the 3 sets of configurations).
The conclusion would be to use the first SMAC run configuration (optimizing enclosing square) until the optimal value for enclosing square and then use the second run configuration (optimizing the overlapping area). Thus we obtain a much better Pareto front than our initial guess.

III. Algorithm Prediction

For the algorithm prediction we decided to use the data from the first section (runs on larger instances) to train a regression model that would predict the amount of time our greedy approach will take to find a solution.
The features are the optimal value and different statistics about the list of square sizes (mean, standard deviation, percentiles, kurtosis, skewness, minimum, maximum, length).
The first regression method was ridge regression. The below table shows the mean root mean square error for different values of alpha parameter (regularization factor) over 5 cross validation folds. The minimum error is 23.83.



We then considered XGBoost with the hope of finding a better result but unfortunately we were not able to fine tune the algorithm to find a better prediction.


test-rmse-mean
train-rmse-mean
60
55
50
45
40
0
1
2
3
4
5
6
7
8
9