VILNIUS UNIVERSITY

FACULTY OF MATHEMATICS AND INFORMATICS

MODELLING AND DATA ANALYSIS MASTER'S STUDY PROGRAMME

Master's thesis

Evaluation of Efficiency of Random Search Optimization Algorithms

Modestas Tamulevičius

Supervisor: **Doc. Dr. Algirdas Lančinskas**

Vilnius, 2022

Contents

Abstract

An application is created for quick and no coding knowledge required analysis of algorithms for optimization problems. Applications building process and data formatting is explained. An overview of two advanced statistical features are explained and involved. Some evolutionary algorithms are compared to showcase applications capabilities and features.

Introduction

In this paper one of the packing problems (class of optimization problems in mathematics) called the Container Loading Problem (CLP) is discussed. This topic is an active field of research that is very closely related to real world applications, usually in container transportation of any kind. It involves sorting rectangular boxes in a substantial rectangular container in a certain way (sequence) so the filled up volume or weight of, for example, a ship is maximized. Latest research [1] strives finding the best double-objective solution so both weight and volume are taken into consideration. Although this seems to be a pretty straight forward task many practical difficulties may arise [4]. For instance, in what orientation are the boxes put (some containers can or cannot be rotated), various limitations (putting heaviest boxes at the sides of a container can cause in-balance), separate sequence of shipments scheduled order might be required.
This type of task is bounded by random number generation as single iteration conclusions cannot be made. In a single experiment thousands of iterations are done in the search of an optimal solution. Many algorithms may be used to find the best sequence. Some of the examples are: the dumb classical pure random search (PRE), promising genetic algorithm (GA) [3], weak to local minimas Multistart (MS) [2] algorithm and more (not to forget that evolutionary models can be widely modified). Articles are being created currently to investigate various features of these models and distinct situations in which they are put to test [4]. As the algorithms are modified and created some sort of comparison and analysis is needed. Some algorithms tend to be based on luck - will they find a local minimum and improve slowly or will they skyrocket to an optimal solution instantly? Which one reaches the solution quicker? Which one deviates the most? How many experiments were worse

than a selected experiment? It's time consuming to use a command line (Jupyter notebooks, MATLAB, etc.) each time one wants to evaluate progress of algorithms. A very handy solution would be a graphical user interface (GUI) application (APP) with which other contributors to this research field could examine different container sequence building methods. Such GUI could implement multiple algorithm containing graphs with classical statistical insights like mean, standard deviation, confidence intervals, etc.. Also some deeper features can be used: probability to find an optimal solution with a certain error and amount of solutions that are worse than a chosen experiment.
First of all, an utility has to be created from a scratch. A clear description of applications input has to be made, because some of the experiments end up failing and so, such inconsistencies in data must be handled with no issues. Secondly, mentioned advanced analysis features have to be described. Not to forget - a great GUI has to be decorated with easy on eyes graphics. Lastly, some of before mentioned evolutionary algorithms will be inspected to give a "live" insight on how the application helps select a smarter technique for a different CLP situation.
Hypothetically, APP should let a user (with knowledge in optimization problems field) realise CLP algorithm performance quickly and easily.
To start with, creation of application from nothing will be recounted. All particularities like format of input files, settings of graphical output (and other choices) will be demonstrated and reasoned. A guide will be provided on how to use such program. Then, some of the statistical features will be developed. By the end of this article, a showcase of few algorithm experiments will conclude this app fully.

All in all, a GUI application is required with different statistical features (at least one advanced feature) used to examine optimization algorithms. Advanced features are going to be developed and described during the process. Program build process, tools used and ways toward utility realisation have to be explained. After the program is present an analysis will be made for 2 pairs of optimization algorithms to showcase analysis of APP. GUI should be understandable, tested and APP should be usable with no command line execution or coding experience.

Python 3.8.5 with additional libraries will be used to achieve this goal.

Literature Overview

The main motivation to develop an analysis application comes from previously done work [2] when comparing abundance of algorithms. To do so, dedicated research specialist has to get used to either command line or some sort of programming language. Why shouldn't one create a tool that doesn't require experience with the code, just some basic packages to be installed on current operating system and the utility itself in .exe format? As seen in this work, benchmarks for various iteration times per experiment were made. Most importantly, this paper inspires the Cumulative Density of the Best Point Found (CDFR) function:

$$\mu(y) = P\{f(x) \leqslant y\}$$

as cumulative distribution function of $y = f(x)$ where x is uniform over X.

Basically, it shows the probability that a level y is reached after generating N trial points:

$$P_N(y) = 1 - (1 - \mu(y))^N$$

CDFR feature gives great overview on how the solutions of algorithms are skewed, distributed, what is the probability to reach a certain threshold solution (or better). This feature will be implemented in APP.

Another feature that is useful for a researcher and is applied in APP is probability to determine the optimal solution with different error levels [4]. This helps a researcher understand to what error (that are always met in real world application) a model is usable. Also, this paper concludes one of the most favourable models: GA. This model combines few best solutions per experiment by splitting each solution into parts and combining with another one. It has some Darwin's Theory of Evolution sense to it hence the name: Genetic Algorithm.

Even though this work doesn't focus on multi-objective problems [1], there are claims that such models are hopeful. This might be another great step for this analysis utility to be upgraded.

As of evaluation heuristics for double-objective problems and most recent results [3], some papers were written in past decade. G.Miranda et al. notes major troubles of CLP optimization in real world: different angles of container, problem with double-objective solutions, etc.. This comes in great synergy with the [4] probability to achieve optimal solution versus error graph. This previously mentioned metric provides a surmountable help realising to what extent a specific model might be applied.

Application development

Functionality requirements

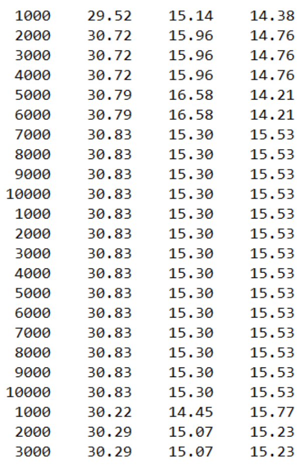
To begin with, let's recount on what the acceptance criteria for an APP is:

- A utility that doesn't require any code to be executed
- Graphical visualization interface of at least one higher complexity feature
- Ability to compare at least 2 algorithms
- Common sense: easy to understand GUI, no bugs, etc.

File format

Further on, files that are being read and analysed come in such form:

- First number (column) in a row represents the amount of iterations made in an experiment, after a space second number shows the best value(solution) algorithm reached after former number of iterations in a particular experiment
- Second row presents the next number of iterations in that same experiment and the best value it reached after this number of iterations (which has increased, and most probably, the reached value has improved)
- Any amount of iterations per experiment is acceptable, but for the current version of program it's required to have values every 1000 iterations and experiment should consist of at least 10000 iterations.
- Second value (column) can be anything - either percentage of total volume or total weight of a container ship loaded up. Actually, it can be any real number value that is needed to be maximized, so any sort of algorithm in these optimization problems can be tested.



```
1000     29.52    15.14    14.38
2000     30.72    15.96    14.76
3000     30.72    15.96    14.76
4000     30.72    15.96    14.76
5000     30.79    16.58    14.21
6000     30.79    16.58    14.21
7000     30.83    15.30    15.53
8000     30.83    15.30    15.53
9000     30.83    15.30    15.53
10000    30.83    15.30    15.53
1000     30.83    15.30    15.53
2000     30.83    15.30    15.53
3000     30.83    15.30    15.53
4000     30.83    15.30    15.53
5000     30.83    15.30    15.53
6000     30.83    15.30    15.53
7000     30.83    15.30    15.53
8000     30.83    15.30    15.53
9000     30.83    15.30    15.53
10000    30.83    15.30    15.53
1000     30.22    14.45    15.77
2000     30.29    15.07    15.23
3000     30.29    15.07    15.23
```

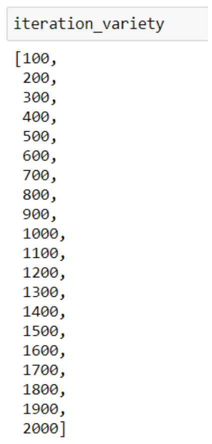
Figure 1: Example of data file

- After rows of a single experiment are finished another one may be stated again in the same manner beginning right from the next row. Of course, for APP to show great conclusions a file with as much experiments as possible is recommended to use.
- It doesn't matter if an experiment has occurred that failed or was stopped before reaching the set maximum number of iterations. APP will throwaway all the inconsistent experiments and leave only full successful ones.
- File has to be in .csv, .txt or .dat formats
- File name is set as models name, any title will be digested by APP as long as APP using researcher has easy time reading it from analysis graphs legends

An example of how the file can look like is present in Figure 1. First column shows increasing iterations from 1000 to 10000 by 1000 steps. Second column shows models best solution after the corresponding amount of iterations. All other columns doesn't make a difference if they exist or not.

Data preparation

For data clean-up and preparation for analysis, *numpy* and *pandas* libraries were used with one of the most common programming languages executed in data modelling: Python (ver. 3.8.5). For current version of APP two algorithm iteration files are imported. After reading formerly described file APP takes out unnecessary



```
iteration_variety
[100,
 200,
 300,
 400,
 500,
 600,
 700,
 800,
 900,
 1000,
 1100,
 1200,
 1300,
 1400,
 1500,
 1600,
 1700,
 1800,
 1900,
 2000]
```

Figure 2: List from a file containing experiments with 2000 iterations every 100 steps

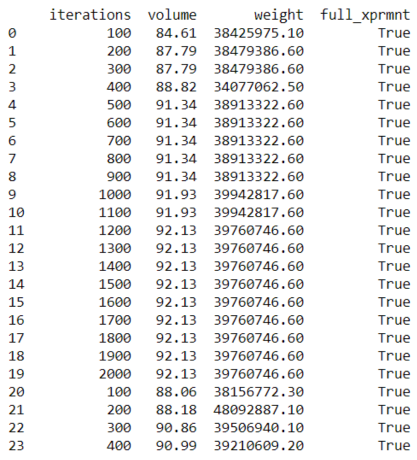
columns. It removes every other column except for the first two and renames them to *"iterations"* and *"value"*. By converting iteration column to integer type, using *.unique()* functionality and sorting a full list of iteration steps per full experiment in current file is provided (see Figure 2) Then, APP finds the full iteration pattern of an experiment and filters out all the incomplete experiments. This is made by using such line of code:

```
data.rolling(window=N , min_periods=N)
                     .apply(lambda x: (x==pat).all())
                     .mask(lambda x: x == 0)
                     .bfill(limit=N-1)
                     .fillna(0)
                     .astype(bool))
```

.rolling() with *.apply()* functions do the main work here - they let APP filter a *pandas* data-frame by full list pattern (Figure 3). These data preparation steps are done for both data files of two algorithms. After both algorithm data-frames are prepared, they get merged with algorithm labels as third column set into a common data-frame (DF from here, see Figure 4).

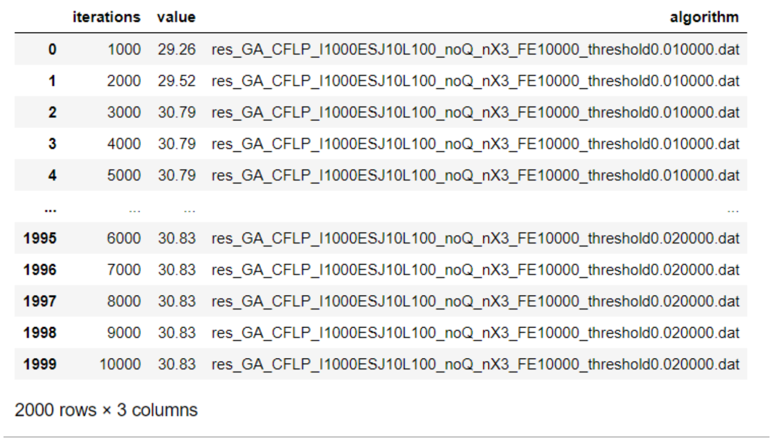
Afterwards, first important variable is set - optimal solution. In the starting version APP best achieved value of both files is set as the optimal solution. This will be useful to project in graphs as the asymptotic line to which algorithms converge, see Figure 5.

For DF another pair of data-frames is created to find minimum and maximum



```
    iterations  volume       weight  full_xprmnt
0          100   84.61  38425975.10         True
1          200   87.79  38479386.60         True
2          300   87.79  38479386.60         True
3          400   88.82  34077062.50         True
4          500   91.34  38913322.60         True
5          600   91.34  38913322.60         True
6          700   91.34  38913322.60         True
7          800   91.34  38913322.60         True
8          900   91.34  38913322.60         True
9         1000   91.93  39942817.60         True
10        1100   91.93  39942817.60         True
11        1200   92.13  39760746.60         True
12        1300   92.13  39760746.60         True
13        1400   92.13  39760746.60         True
14        1500   92.13  39760746.60         True
15        1600   92.13  39760746.60         True
16        1700   92.13  39760746.60         True
17        1800   92.13  39760746.60         True
18        1900   92.13  39760746.60         True
19        2000   92.13  39760746.60         True
20         100   88.06  38156772.30         True
21         200   88.18  48092887.10         True
22         300   90.86  39506940.10         True
23         400   90.99  39210609.20         True
```

Figure 3: Extra column is assigned that shows if the subsequent rows are from a full experiment



	iterations	value	algorithm
0	1000	29.26	res_GA_CFLP_I1000ESJ10L100_noQ_nX3_FE10000_threshold0.010000.dat
1	2000	29.52	res_GA_CFLP_I1000ESJ10L100_noQ_nX3_FE10000_threshold0.010000.dat
2	3000	30.79	res_GA_CFLP_I1000ESJ10L100_noQ_nX3_FE10000_threshold0.010000.dat
3	4000	30.79	res_GA_CFLP_I1000ESJ10L100_noQ_nX3_FE10000_threshold0.010000.dat
4	5000	30.79	res_GA_CFLP_I1000ESJ10L100_noQ_nX3_FE10000_threshold0.010000.dat
...	...	...	...
1995	6000	30.83	res_GA_CFLP_I1000ESJ10L100_noQ_nX3_FE10000_threshold0.020000.dat
1996	7000	30.83	res_GA_CFLP_I1000ESJ10L100_noQ_nX3_FE10000_threshold0.020000.dat
1997	8000	30.83	res_GA_CFLP_I1000ESJ10L100_noQ_nX3_FE10000_threshold0.020000.dat
1998	9000	30.83	res_GA_CFLP_I1000ESJ10L100_noQ_nX3_FE10000_threshold0.020000.dat
1999	10000	30.83	res_GA_CFLP_I1000ESJ10L100_noQ_nX3_FE10000_threshold0.020000.dat

2000 rows × 3 columns

Figure 4: DF example for GA algorithms that were tested for 10000 iterations per experiment

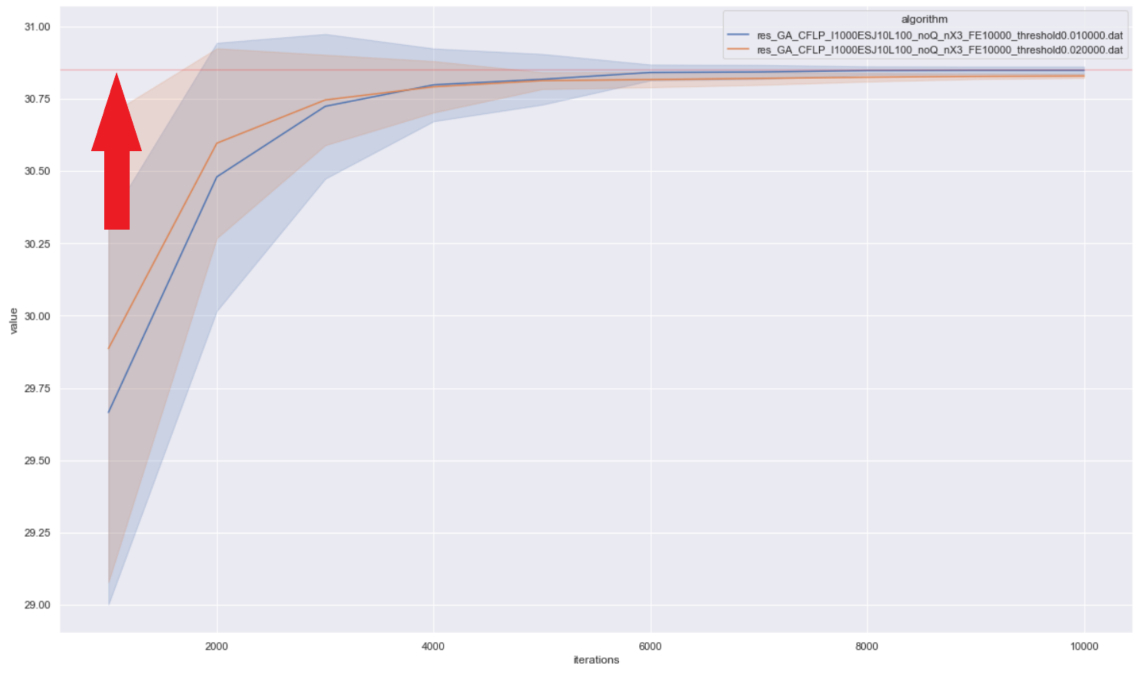
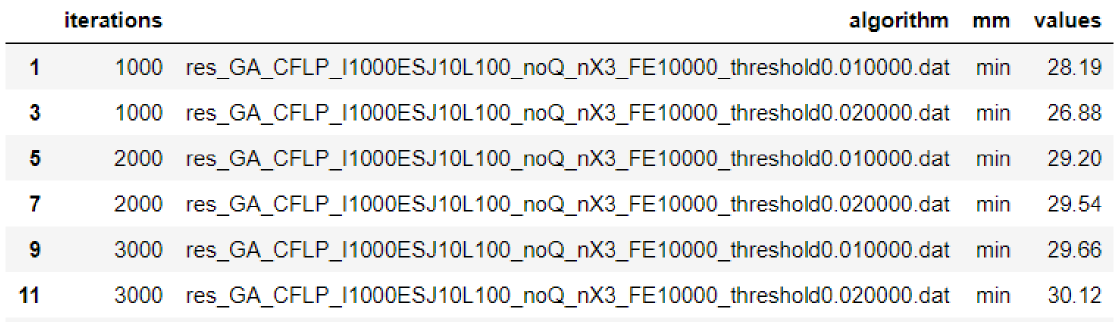


Figure 5: Example of an optimal solution line generated



	iterations	algorithm	mm	values
1	1000	res_GA_CFLP_I1000ESJ10L100_noQ_nX3_FE10000_threshold0.010000.dat	min	28.19
3	1000	res_GA_CFLP_I1000ESJ10L100_noQ_nX3_FE10000_threshold0.020000.dat	min	26.88
5	2000	res_GA_CFLP_I1000ESJ10L100_noQ_nX3_FE10000_threshold0.010000.dat	min	29.20
7	2000	res_GA_CFLP_I1000ESJ10L100_noQ_nX3_FE10000_threshold0.020000.dat	min	29.54
9	3000	res_GA_CFLP_I1000ESJ10L100_noQ_nX3_FE10000_threshold0.010000.dat	min	29.66
11	3000	res_GA_CFLP_I1000ESJ10L100_noQ_nX3_FE10000_threshold0.020000.dat	min	30.12

Figure 6: Example of an optimal solution line generated

values per iteration per algorithm. Min-Max values will be projected in a graph later on (Fig. shows minimum values data-frame for both algorithms: 6).

To present the CDFR calculations had to be made. Cleaned up data-frames of both algorithms are filtered to only include reached values at iteration number equal to N. Then APP counts for each experiment how many other experiments with that algorithm were equal or better than the particular one (APP is set for maximization problems, but is easily coded for minimization). Dividing these counts of experiments by total count of experiment per model probabilities to reach a certain value after N iterations is obtained. Both algorithm probabilities are merged into a single table for later use. In this version of APP 4 values of N in CDFR feature were chosen: 1000, 2000, 5000, 10000, so 4 tables are created for each of mentioned N. In case of two algorithms for same optimization exercise normalization of such feature is not required. If it's a case of same algorithm but for 2 different problems normalization should be done. APP is not normalizing the CDFR feature. Here is a CDFR calculation script:

```
def count_less_than(df1, df2, N):
    max_exp1 = df1.groupby('iterations').count().iloc[0][0]
    max_exp2 = df2.groupby('iterations').count().iloc[0][0]

    N_endings1 = df1[df1['iterations']==N]
    N_endings2 = df2[df2['iterations']==N]

    N_endings1['probability']=0
    N_endings2['probability']=0
    for index, row in N_endings1.iterrows():
        N_endings1.at[index,'probability'] =
            N_endings1[N_endings1['value']>=row['value']]['value'].count()

    for index, row in N_endings2.iterrows():
        N_endings2.at[index,'probability'] =
            N_endings2[N_endings2['value']>=row['value']]['value'].count()

    N_endings1['probability']=N_endings1['probability']/max_exp1
```

```
N_endings2['probability']=N_endings2['probability']/max_exp2


df = N_endings1.append(N_endings2)
df = df.reset_index()
return df.drop(columns='index')
```

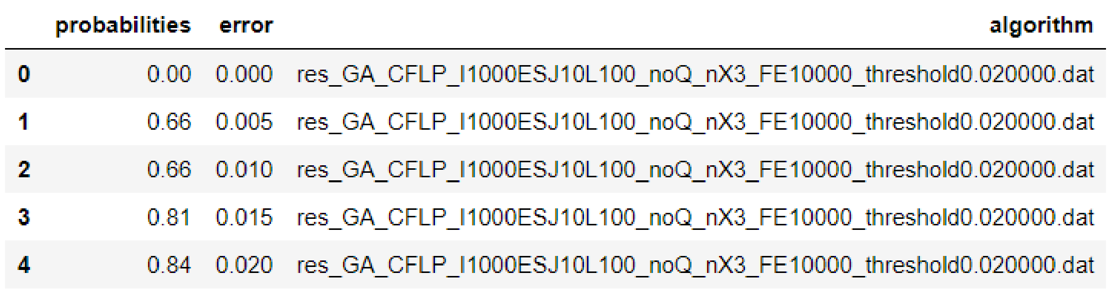
Here, for example, *max_exp1* represents count of experiments in the first files cleaned up data-frame (FF) (first file in a sense that was put into the *count_less_than()* function), *N_endings1* is FF filtered to only contain a selected N iteration steps. For example, only results after 1000 iterations for each experiment. "for loops" run through both data-frames searching for values that are higher or equal to the current row. After that "probability" columns need to be divided by total number of successful experiments per each file. This is it, we have defined $CDFR_N$.

Lastly, probabilities to reach optimal solution with a certain % of error for a specific N of iterations per experiment are calculated. This is pretty straightforward - both algorithm data-frames are filtered to only include N iteration values (similarly to $CDFR_N$ case) and only those that are better or equal to the optimal solution with certain error are counted. These counts are then divided by total number of experiments to get probabilities:

$$P_N(X \geqslant op\_solution * error) = count_N^{x \geqslant op\_solution * error} / total\_experiment\_count$$

where x is reached values of an algorithm, N iteration step selected per experiment.

Error levels of {0, 0.005, 0.01, 0.015, 0.02} and same N values (as for CDFR feature) were taken into consideration. An example of probability error dataframe : Figure 7



	probabilities	error	algorithm
0	0.00	0.000	res_GA_CFLP_I1000ESJ10L100_noQ_nX3_FE10000_threshold0.020000.dat
1	0.66	0.005	res_GA_CFLP_I1000ESJ10L100_noQ_nX3_FE10000_threshold0.020000.dat
2	0.66	0.010	res_GA_CFLP_I1000ESJ10L100_noQ_nX3_FE10000_threshold0.020000.dat
3	0.81	0.015	res_GA_CFLP_I1000ESJ10L100_noQ_nX3_FE10000_threshold0.020000.dat
4	0.84	0.020	res_GA_CFLP_I1000ESJ10L100_noQ_nX3_FE10000_threshold0.020000.dat

Figure 7: Example data-frame generated for Probability-Error analysis

Application development

Application is set in functions and class objects. For data preparation and visualization functions were made, but for GUI windows class objects were used. *tkinter* library with *matplotlib* came in handy to create the classical windowed program. Classes were made to represent overall *tkinter* windowed program and each displayable *tkinter* page that could be opened. Buttons and functions were assigned with the corresponding page, for example, here's a code snippet to reach "Mean-SD" graph from the StartPage Class:

```
class StartPage(tk.Frame):


    def __init__(self, parent, controller):
        tk.Frame.__init__(self,parent)
        label = tk.Label(self, text="Start Page", font=LARGE_FONT)
        label.pack(pady=10,padx=10)


        button = ttk.Button(self, text="Mean-SD",
                            command=lambda:
                                controller.show_frame(Mean_SD_Graph))
        button.pack()
```

Graphical analysis solutions were solely made by *seaborn* library that provided greater prosperity than it's core library *matplotlib.* Each data-frame is formed in such manner, to use as simple graph drawing functions as possible. Mainly *.lineplot()* and *.scatterplot()* were used to provide great visual interface. "hue" setting inside, for instance, *seaborn .lineplot()* function allows to quickly assign visually attractive and separable colours per group (in this case, per algorithm). *.axhline()* modification to lineplots provided an easy way to showcase optimal solutions. APP provides four main categories of graphs: Mean-SD, CI-MIN-MAX, CDFR and Probability-Error. Mean-SD graph shows average value and standard deviation per iteration step per algorithm. Second category shows confidence intervals of 95% with minimum and maximum values per iteration step per algorithm. Last two categories visualize features described in Data preparation subsection. Each of these categories are divided into {1000, 2000, 5000, 10000} N value graphs.

pyinstaller library formed APPs Python code from a .py file to a single executable (.exe file).

Analysis of algorithms

Let us test APP for few different algorithms to see what it actually has to offer.

Genetic Algorithm [2] has a logic of initial set of M items $\mathbb{I} = \{\mathbb{A}_1, \mathbb{A}_2, ..., \mathbb{A}_M\}$ which are initially generated randomly. This set of items is then manipulated with three main arguments: crossover, mutation and selection. Crossover pairs two items

$$\mathbb{A}^{(1)} = \{a_1^{(1)}, a_2^{(1)}, ..., a_n^{(1)}\}$$

$$\mathbb{A}^{(2)} = \{a_1^{(2)}, a_2^{(2)}, ..., a_n^{(2)}\}$$

Which are randomly chosen from $\mathbb{I}$, to generate a new offspring item $\mathbb{A}^{(3)}$. Part of this offspring is made up from first $\mathbb{A}^{(1)}$ and rest is made up from $\mathbb{A}^{(2)}$. This crossover has probability π_c to occur. If it didn't occur, one of the "parents" will fully make up this new offspring $\mathbb{A}^{(3)}$. It's important to note that GA is hard to analyse as it's one of the evolutionary algorithms - they are known for having insane amount of parameters to set - allele, genome, SotF and many others. For example, GA module in popular mathematical analysis package MATLAB has 26 arguments out of which 17 are manipulating performance.

In first case two versions of GA will be used - one with threshold arguments set to 0,02 and another one with 0,01. Let us run through full list of APPs graphical solutions. First comes Mean and Standard Deviation (SD) Figure 8. Mean and SD are pretty obvious statistical insights - clearly GA model with threshold 0,01 (GA0.01) threshold setting is performing worse than the GA with threshold 0,02 (GA0.02). At the beginning GA0.01 performs worse, but at the 4000 iterations it overtakes the GA0.02 by small amount ($\leqslant 0,1$ of value). Light red line portrays optimal solution (best solution which is reached by GA0.01). Standard deviation at 1000 iterations is larger for GA0.02, but later on GA0.01 starts deviating hard until the stop at 6000 iterations. Clearly, threshold of 0.01 for GA is showing better performance.

From Confidence Intervals with Minimum-Maximum values graph Figure 9 it seems that both models show similar confidence intervals at 95%. Both models

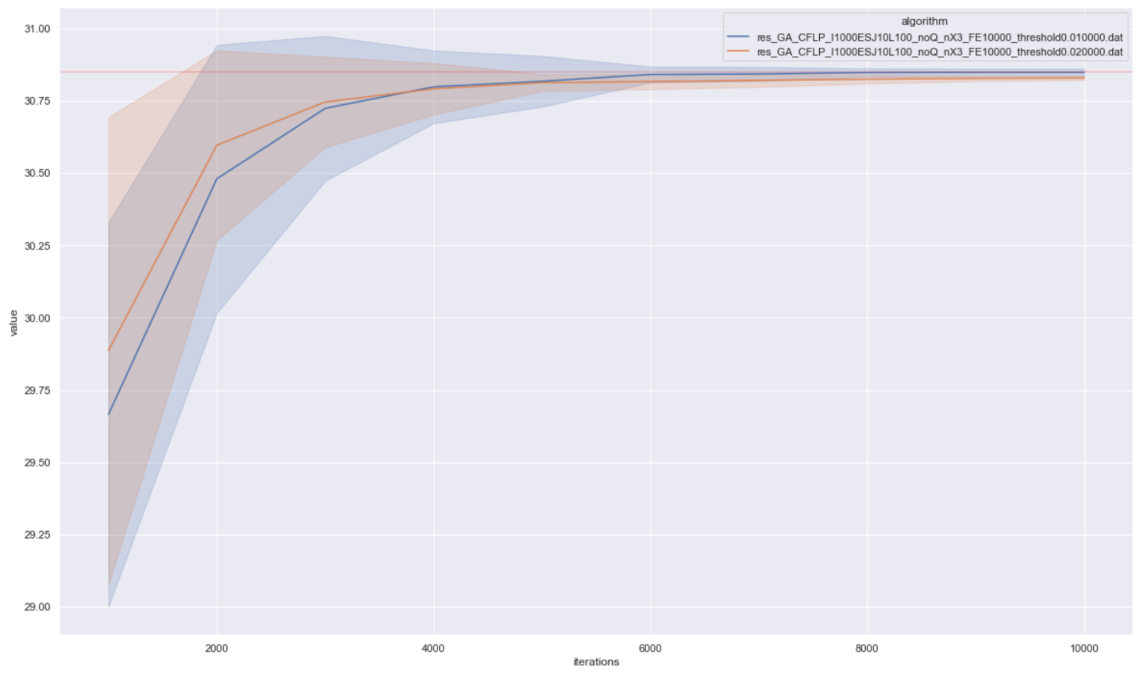


Figure 8: Mean - Standard Deviation graph

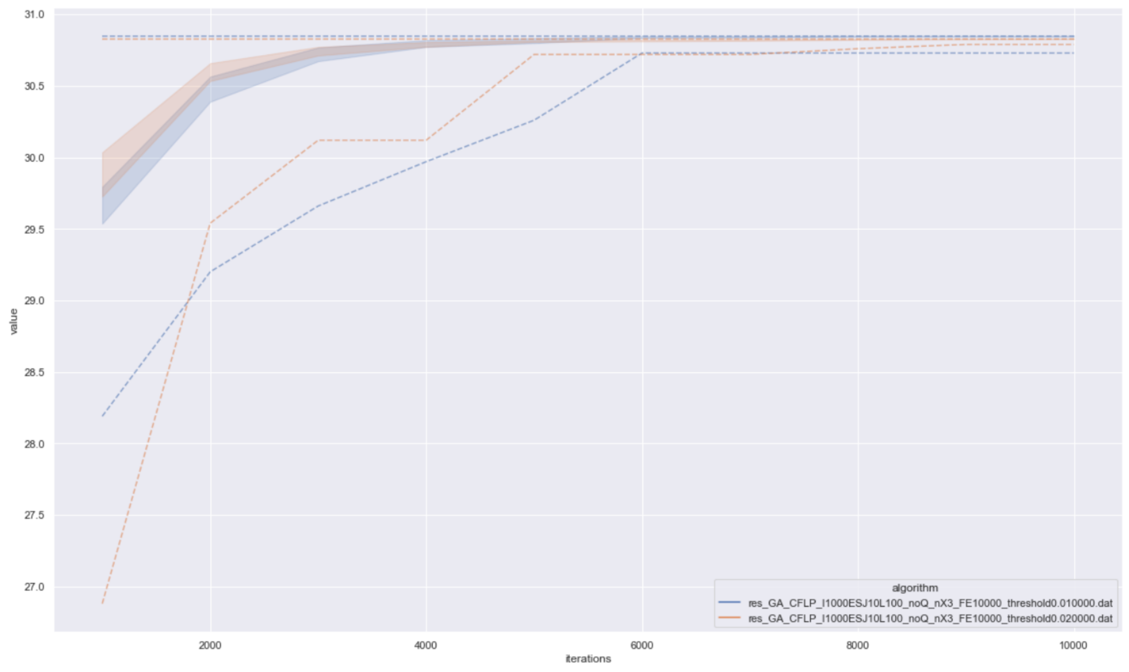


Figure 9: Confidence Intervals of 95% with Min-Max values

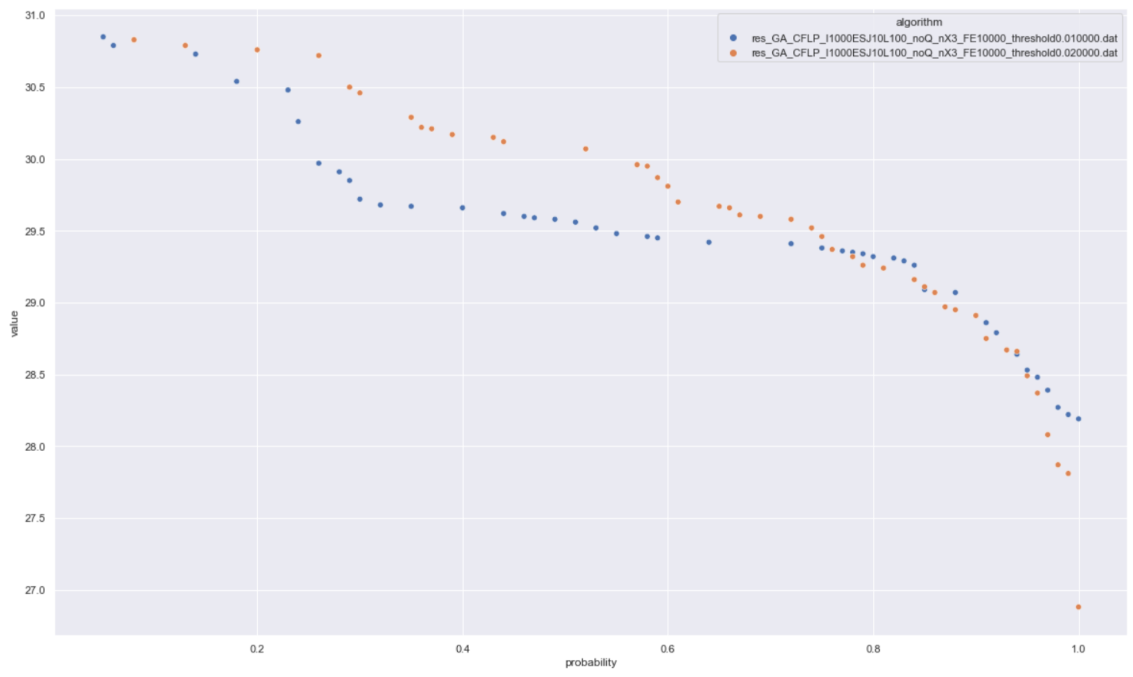


Figure 10: $CDFR_{1000}$

show that they can reach the best solution in 1000 iterations and then fail to improve it for 10000 iterations. Minimum marked line might give a great idea of how worst case scenario experiment looks like for each a algorithm, both of algorithms seem to become equivalent after 6000 iterations.

$CDFR_{N=1000}$ graph in Figure 10 gives interesting insights: firstly GA0.02 seems to outperform GA0.01, but this might be misleading. Higher values have greater probability (P) for GA0.02 - they are higher than the "blue" counter parts. GA0.01 seems to have sudden drop in probabilities to meet high values (down from 30,5). As probability rises at around 0.8 GA0.02 suddenly drops in values. If we analyse only from that range (P $\geqslant 0,8$) we see a very slight GA0.01 improvement over GA0.02. Lastly, close to probability 1, large GA0.01 leadership in values is seen.

$CDFR_{2000}$ graph in Figure 11 gives a clearly different angle on the thresholds. After 2000 iterations (reminder: N setting) GA0.02 has it's "dots" gathered majority of it's values at high probabilities. It shows smaller deviation and better probability for great results. On the other hand, GA0.01 has lower probability to obtain same values as GA0.02. Also, it's interesting to see GA0.02 drop in values at around $[0,7;0,8]$ probabilities. Overall decrease in values with $P \leqslant 0,75$ shows that both models with the growth of N tend to have higher chance acquiring great solutions.

For $CDFR_{5000}$ in Figure 12 and $CDFR_{10000}$ in Figure 13 show only probability range from 0,7 and 0.97 respectfully and very few value points which just shows that

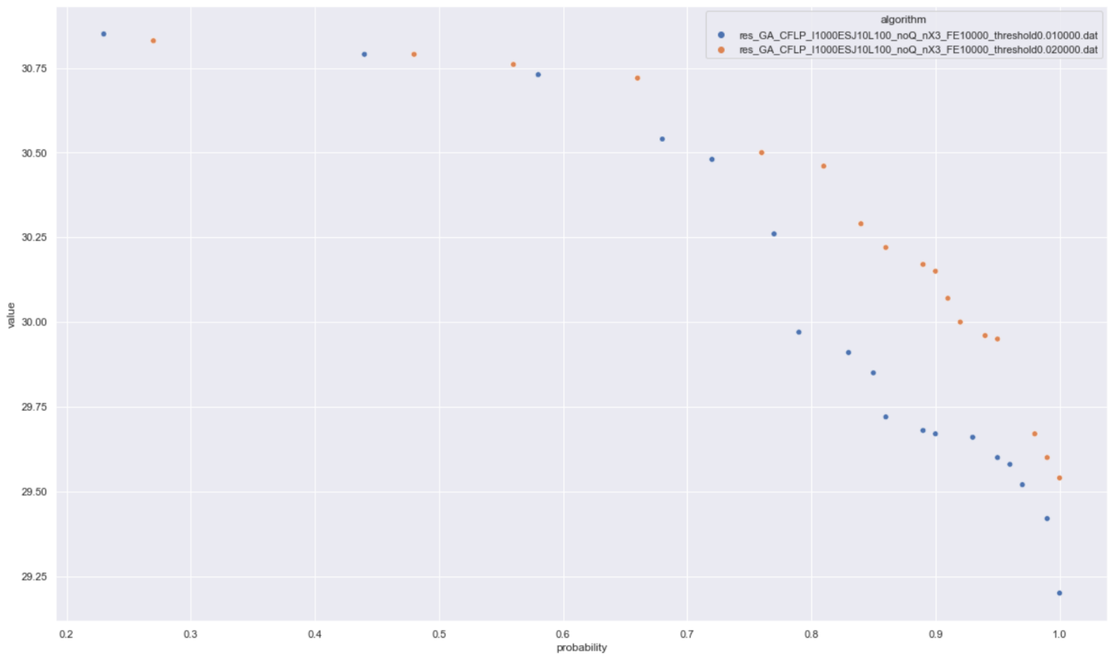


Figure 11: $CDFR_{2000}$

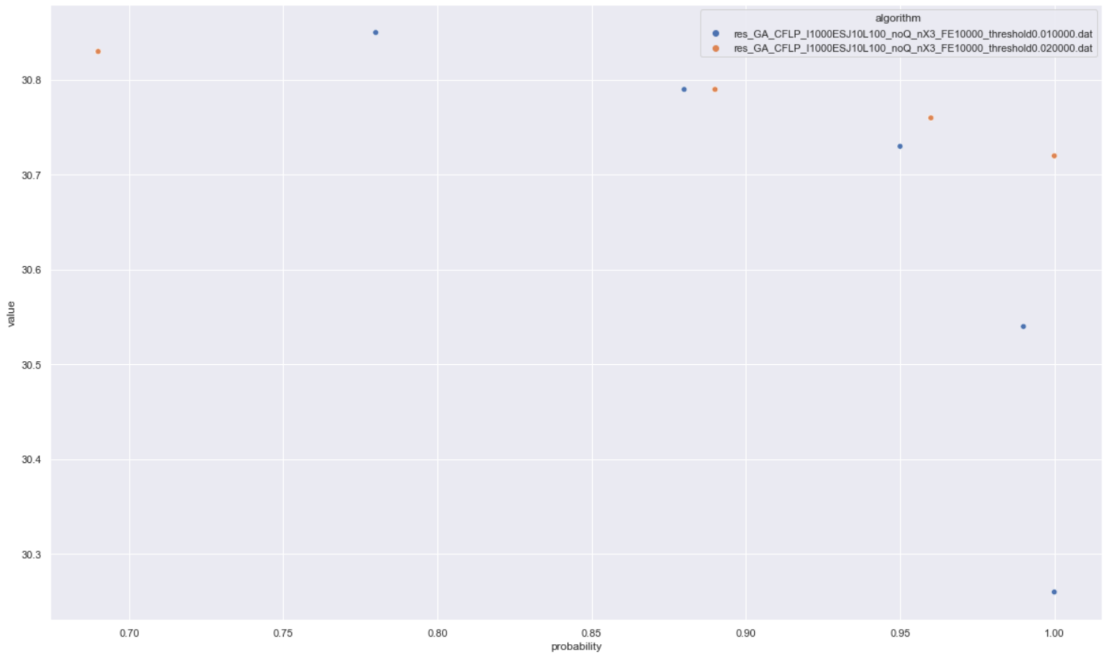


Figure 12: $CDFR_{5000}$

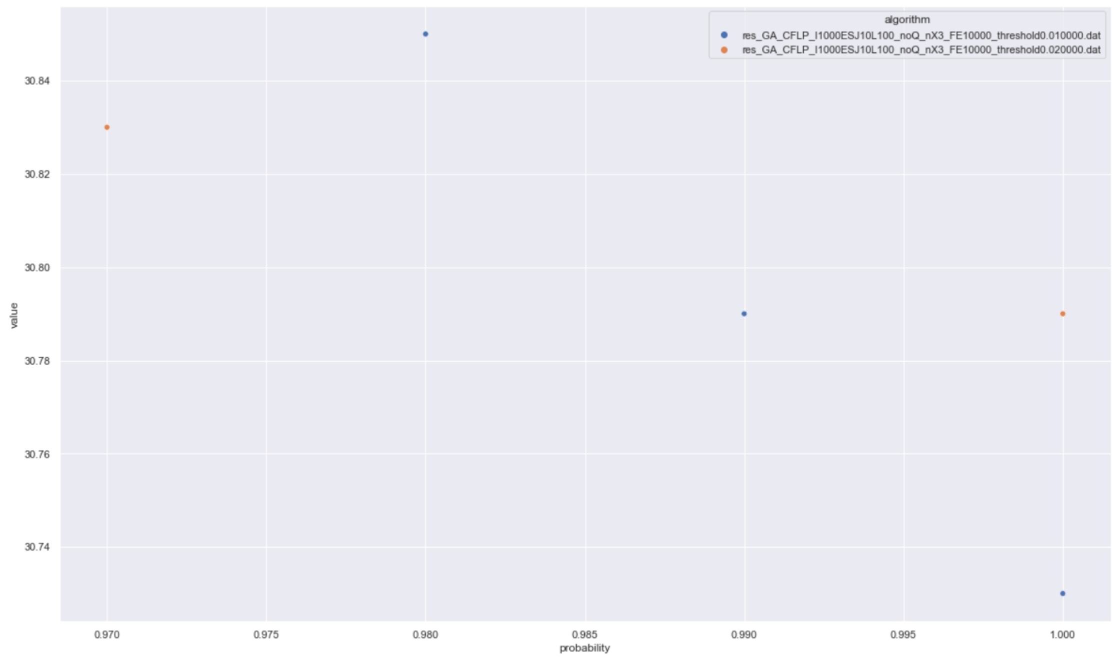


Figure 13: $CDFR_{10000}$

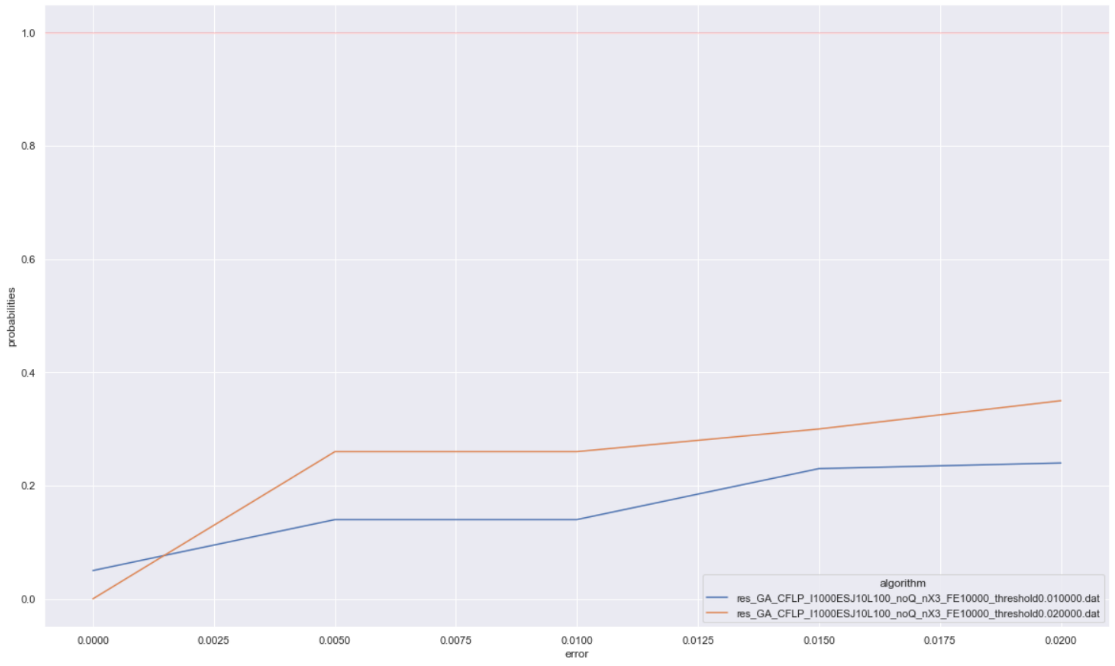


Figure 14: $Probability_{N=1000} - Error$

great conclusions cannot be made for these models at high N.

Moving on to the last feature - probability to reach an optimal solution with certain errors at N=1000. In Figure 14 - GA0.02 at almost all error rates wins over GA0.01, only at no error rate GA0.01 has a slight chance of $P \leqslant 0,075$ to find an optimal solution.

Figure 15 shows same graph for N=2000. Both GA threshold settings promise a

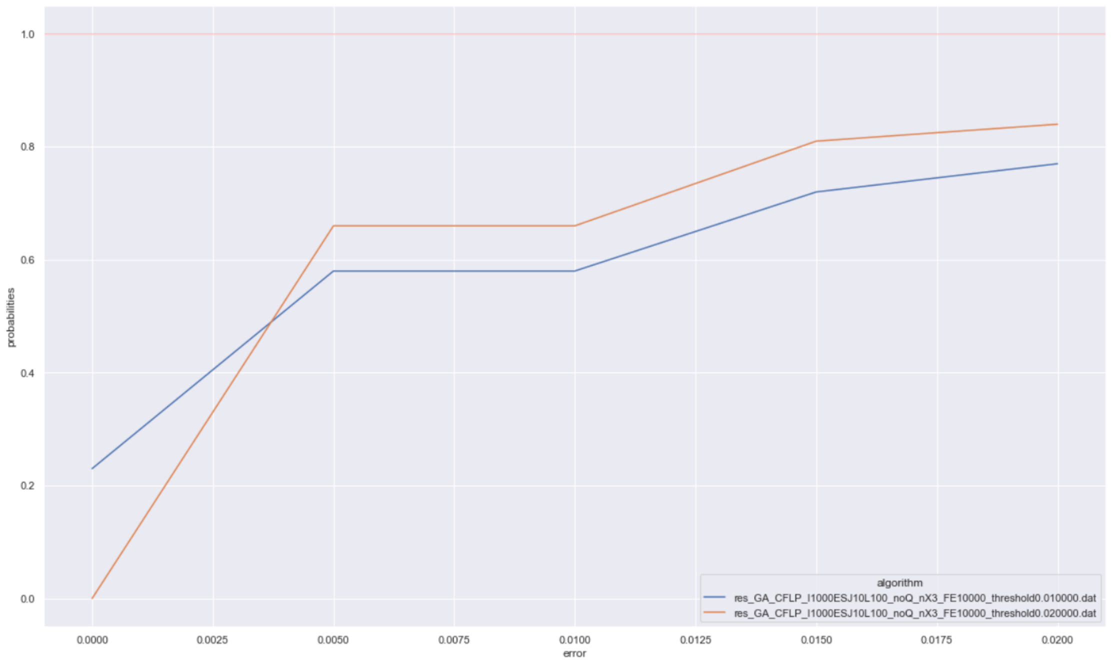


Figure 15: $Probability_{N=2000} - Error$

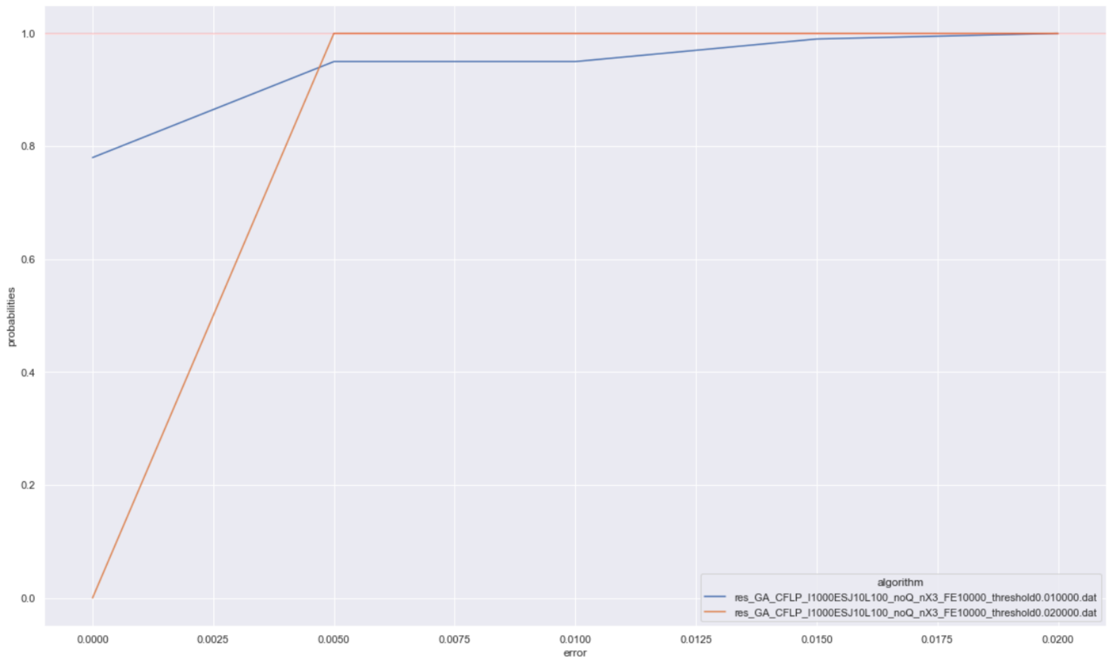


Figure 16: $Probability_{N=5000} - Error$

lot, but GA0.02 overcomes GA0.01 once again.

Figures 16 and 17 bring no information as the optimal solution at N=5000,10000 was reached with almost any error level.

Overall, GA threshold setting at 0,02 consistently showed better results, and probabilities at error=0 can be dismissed - let us remember how we described optimal solution: it is max value reached by any of two algorithms in any experiment

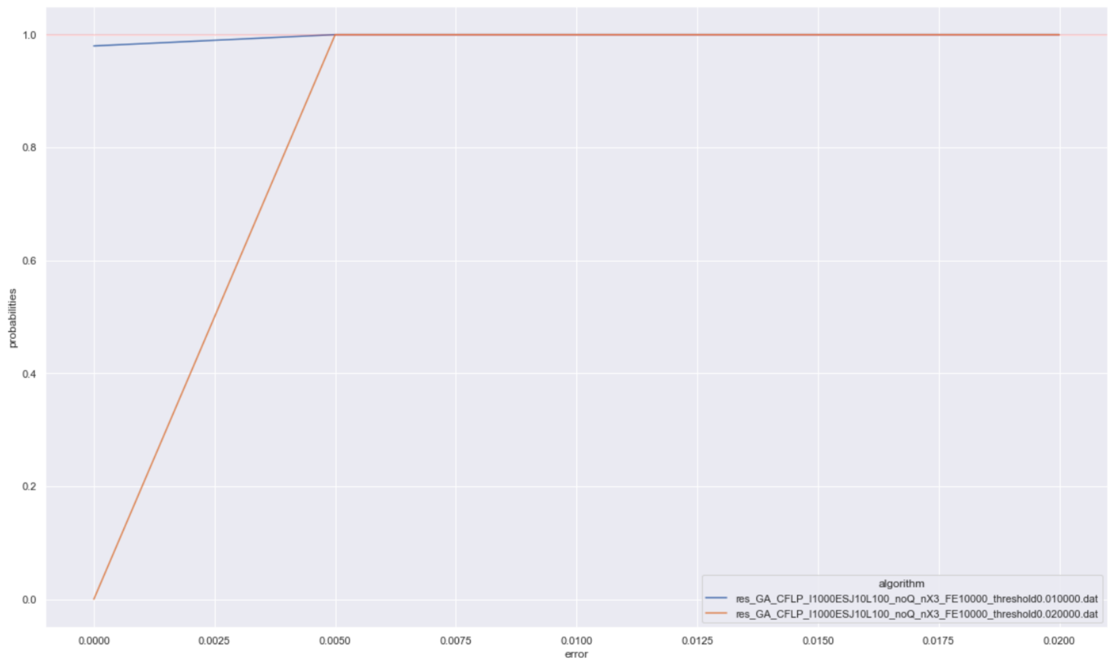


Figure 17: $Probability_{N=10000} - Error$

- GA0.01 reached it and GA0.02 didn't. Even though GA0.01 performed worse iteration-wise and in majority of other senses, it still managed to reach a slightly better solution.

For a second experiment let's test PSO and Multi-Start (MS) algorithms in a minimization problem with both model dimension set to $d = 2$. As data was available for 1000 iterations, let's inspect $CDFR_{N=1000}$ and probability vs. error graph only.

PSO model [2] is a different evolutionary method where swarm intelligence and cognitive consistence terms arise. Here complex sorting pattern for each "swarm" of population is used with a newly coined term velocity, which shows how much a swarm changed at random.

Multi-Start [2] is an algorithm which local optimization function $LS(s, N)$ is used. It's given a starting point S with limit N (function evaluations - same as previously stated N variable). LS then provides a point (approximate minimal solution) in the needed domain.

For $CDFR_{N=1000}$ Figure 18 is monotonically decreasing since it's a minimization problem (Ackley's test problem). It's pretty straight forward that PSO is performing way better in this domain than MS does. As it's a problem of minimization, lower values are searched for. PSO easily provides lesser values at any set probability and is a better choice for Ackley's test.

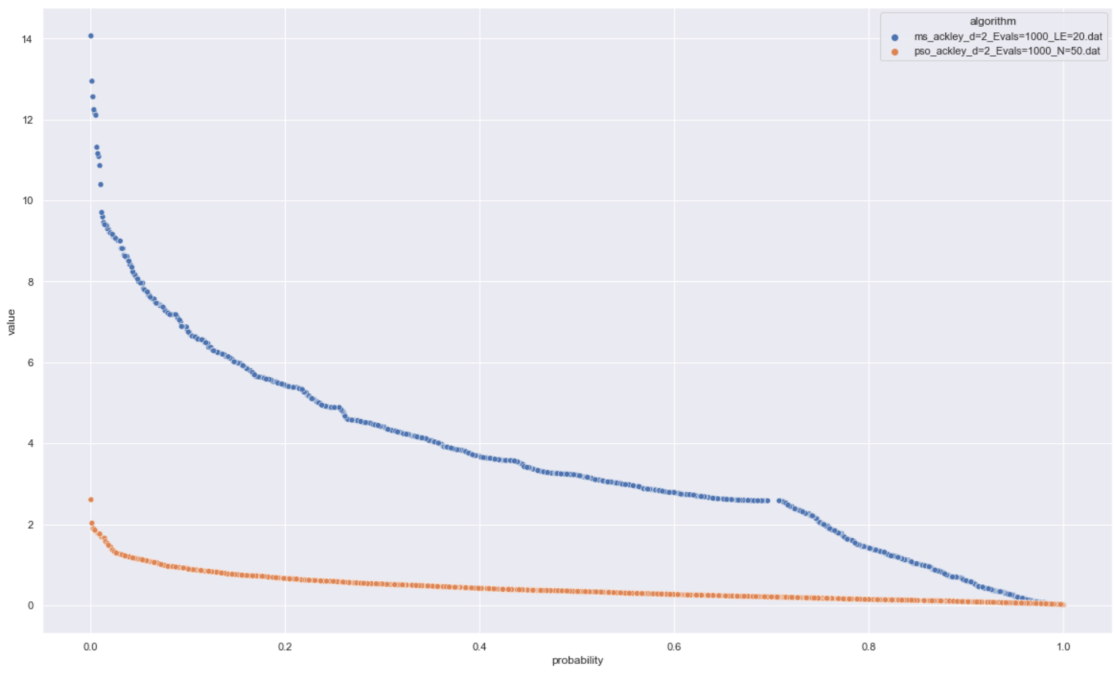


Figure 18

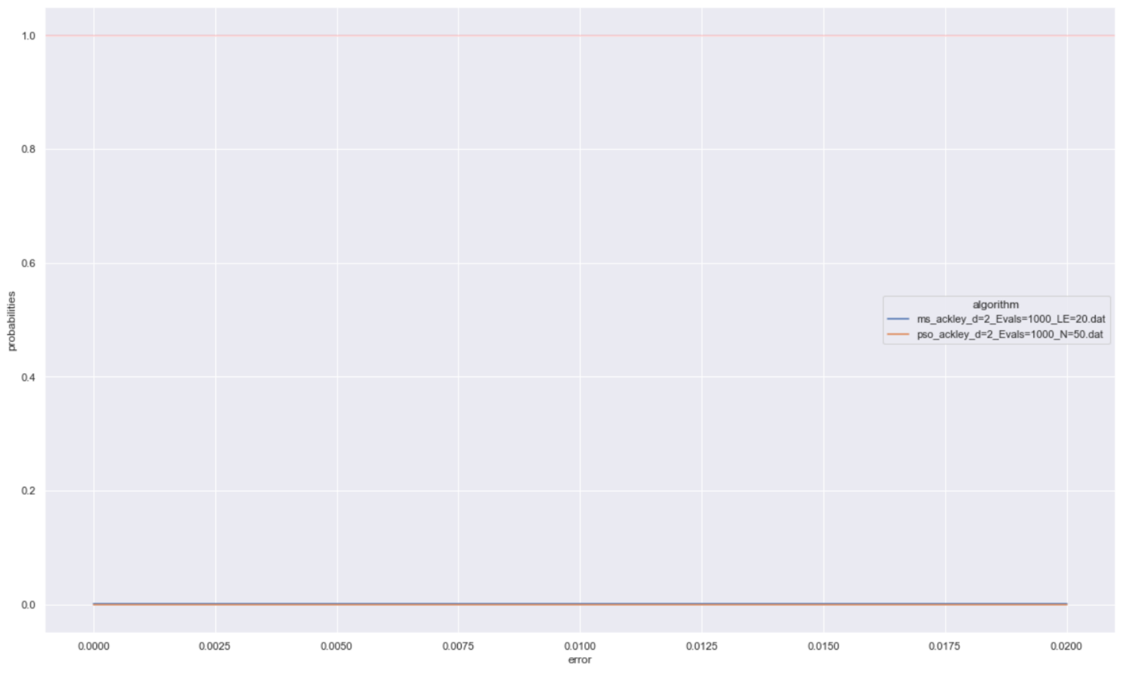


Figure 19

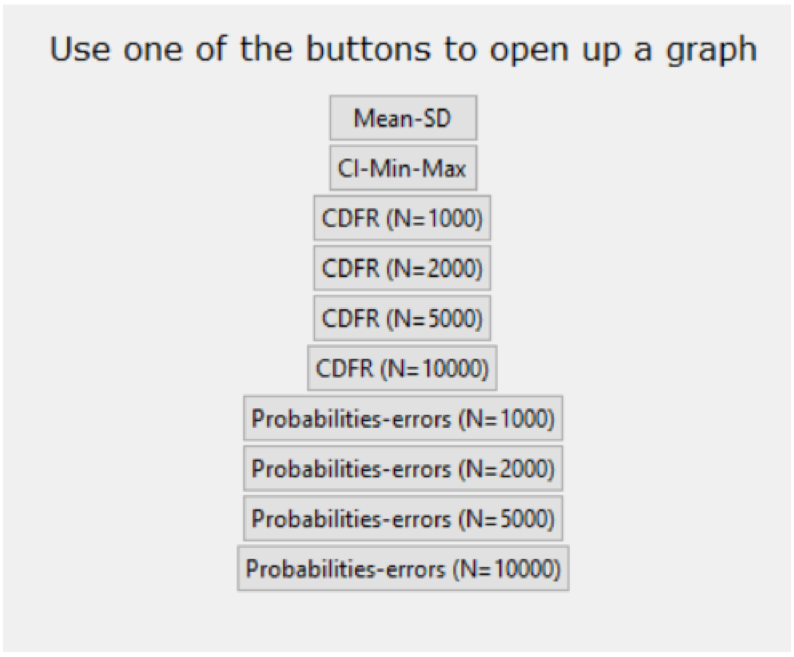

Use one of the buttons to open up a graph
Mean-SD
CI-Min-Max
CDFR (N=1000)
CDFR (N=2000)
CDFR (N=5000)
CDFR (N=10000)
Probabilities-errors (N=1000)
Probabilities-errors (N=2000)
Probabilities-errors (N=5000)
Probabilities-errors (N=10000)

Figure 20

On the other hand Figure 19 shows no useful information - probabilities are so low (P=0,0005) for both models because of high amount of experiments and low amount of iteration count N.

Additional Guidance

There are few tips and tricks when using APP. After launching it user sees the GUI. From there, it's pretty straight forward - one should select a wanted graphical visualisation Figure 20.

Either APP is launched by executing the analysis.py file or by launching the APP.exe file user should only put 2 data files in the same directory as the executable or script.

In each graphs window a toolbar is present to zoom in, zoom out, move around the graph, an option to save a graph to a file is also present. Researcher can move his optical mouse device around the graph to see precise values of a particular visualization.

Conclusion

To sum up, a comfortable to use GUI app was created to help analyse CLP and other optimization field related algorithms. In parallel with the classical statistical insights another few of the features were involved - CDFR and Probability vs. Error Rate. PSO with MS and GA (sub)models were compared. Graphs provided by the app helped examine these algorithms - which ones strike quickly towards optimal solution. As well, how fast do they do so. For example, GA with it's different parameters showed different outcomes - value that should be set in the future to quickly reach better result in the set circumstances were identified (threshold = 0,02). PSO looked more promising over MS in Ackley's test problem. Overall, multiple algorithms showcased with only few clicks optimise the researchers workflow. For further program development, double-objective analysis may be set up. Additional insight metrics might be involved. Additionally, if data-preparation of input files was developed further an almost-all-input-files-embracing application could be launched. Regarding GUI, a major improvement can be made - an automatic iteration steps (N values) drop-down list can be added, so the APP has a more minimalist touch to it.

Other APP information

GitHub link for code inspection - this repository contains full code of APP and has a short guide on how to create a single file executable with pyinstaller library. Newest version of Python (and adding it to WINDOWS PATH) is required for pyinstaller to work. If a reader wishes to tryout APP.exe, please contact the author of this paper, as the executable file takes 300MB+ space and can't be uploaded to GIT environment.

References

[1] G. Miranda; A. Lančinskas; Y. González. Single and multi-objective genetic algorithms for the container loading problem. *GECCO '17 Companion*, 2017.

[2] Eligius M.T. Hendrix; A. Lančinskas. On benchmarking stochastic global optimization algorithms. *INFORMATICA*, 2015.

[3] G. Miranda;Y. González; C. Leon. A multi-level filling heuristic for the multi-objective container loading problem. *SOCO'13-CISIS'13-ICEUTE'13*, 2014.

[4] B. Pelegín; J. Žilinskas; A. Lančinskas; P. Fernández. Improving solution of discrete competitive facility location problems. *Optim Lett*, 2015.