Exact and Genetic Algorithm for TSP

damnko

February 5, 2019

Introduction

The goal of the project was to implement an exact optimization algorithm and a heuristic algorithm of choice for the solution of a specific flavor of the TSP problem.
Both approaches were explored and compared using different constraints in order to identify which was performing better in specific scenarios. Such constraints were primarily time limits, specifically: 0.1s, 1s, 10s, 30s, 80s. Both approaches were also tested under no time constraint. Numerous instances were generated and fed to the optimization algorithm in order to gather sufficient data to guarantee better reliability of the computed statistics.

Source files structure

The project source files are structured as follows:

- `001-generate-points.py`
  Contains the logic for generating the point distributions, the script previews the generated distribution and stores it inside the respective `/points` folders after user approval;
- `002-optimize.py`
  Runs the optimization process through a batch script that calls the OPL solver, this is done on all instances created at the previous step. The OPL model is stored inside the `/opl-model` folder;
- `003-extract-results.py`
  Extracts the results from the files generated by OPL at the previous step and stores them in a convenient way in the `results.json` file;
- The files `genetic_model.py` and `helpersGeneticAlgo.py` contain the implementation of the genetic algorithm using the `deap` python library and some custom functions for *crossover* and *mutation*;
- `004-hypspace-exploration.py`
  Performs the parameter space exploration for the genetic algorithm using the `hyperopt` python library;
- `005-parameter-exploration-analysis.ipynb`
  Is a jupyter notebook performing the analysis on the results of the parameter space exploration;
- `006-optimize-genetic.py`
  Runs the optimization process with the genetic algorithm, this is done on all instances created at the first step;
- `007-genetic-algo-animation.ipynb`
  Generates an animation showing the evolution of the individuals across the various generations of the genetic algorithm;
- `008-analysis.ipynb`
  Is a jupyter notebook performing the analysis on the performances of the exact and genetic algorithms.

1 The model

The mixed integer programming problem to be implemented is the following:

$$
\begin{aligned}
\min \quad & \sum_{i,j:(i,j)\in A} c_{ij} y_{ij} & & \\
\text{s.t.} \quad & \sum_{j:(0,j)\in A} x_{0j} & = |N| & \\
& \sum_{i:(i,k)\in A} x_{ik} - \sum_{j:(k,j)\in A} x_{kj} & = 1 & \quad \forall k \in N \setminus \{0\} \\
& \sum_{j:(i,j)\in A} y_{ij} & = 1 & \quad \forall i \in N \\
& \sum_{j:(i,j)\in A} y_{ij} & = 1 & \quad \forall j \in N \\
& x_{ij} & \leq |N| y_{ij} & \quad \forall (i,j) \in A \\
& x_{ij} \in \mathbb{Z}_+ & & \quad \forall (i,j) \in A \\
& y_{ij} \in \{0,1\} & & \quad \forall (i,j) \in A.
\end{aligned}
$$

Where x_{ij} and y_{ij} are the decision variables and represent respectively the amount of flow shipped from i to j and whether any flow is shipped from node i to node j. The model constraints ensure respectively the following properties:

1. The total amount of flow in the network is equal to the number of nodes
2. The net balance of flow has to be equal to one, so every node has to be visited
3. Only one unit of flow exits from node i, ie. the route exits from node i only once
4. Only one unit of flow enters on node j, ie. the route enters node j only once
5. Limits the amount of flow in case node i and node j are connected

It is worth noting that constraint 3 and 4 won't be sufficient to guarantee a Hamiltonian path, in fact without the subtour elimination constraint 2, subgraphs would be admissible.

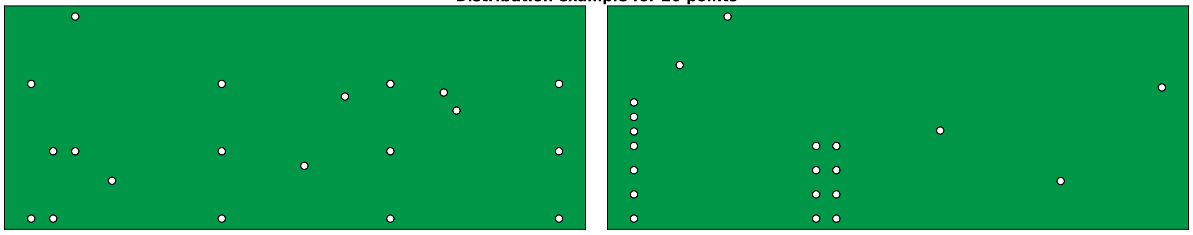
2 Instance creation

In order to have reliable results, it was important to generate input data for the models that represented a good approximation of the problem at hand. Since in this context of the TSP the *locations* are holes in a PCB (electric frame), it was important to generate distributions of points which resembled that scenario.
The file `001-generate-points.py` generates relevant point distributions, using a mixture of the following criterias:

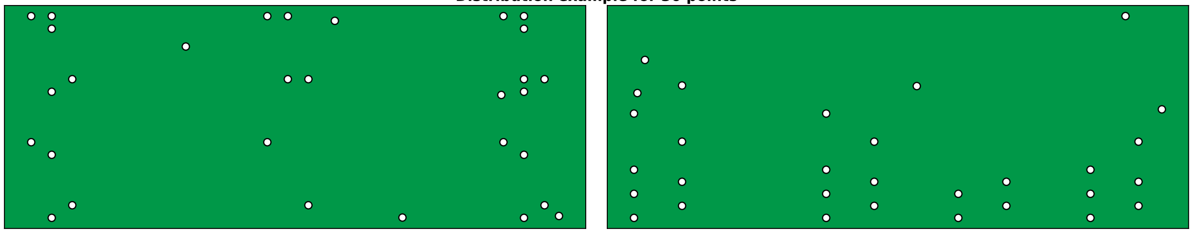
- Distributed at random
- Distributed as a grid
- Distributed as a grid with offset on alternate rows
- Distributed on the vertices of a triangle
- Distributed on a diagonal

After generating a distribution, it is shown to the user for approval before storing it on an external file. The distribution of the points is saved as `numpy` compatible format in `*.npz` files while the distance matrix, computed using euclidean distances, is stored in `*.dat` files.
To have a reasonably big sample of data, distributions for points of size 10, 20, 30, 40, 50, 60, 70, 80 were generated. For each of them, 10 random variations were simulated and saved for future input to the models. So in total 80 different distributions were generated.
Following are some examples of such distributions.

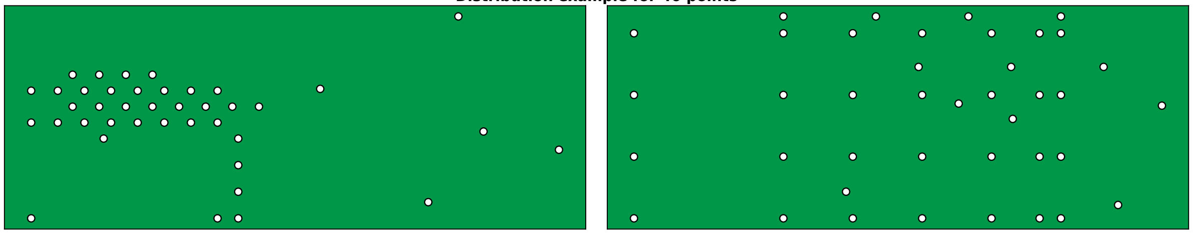
Distribution example for 20 points



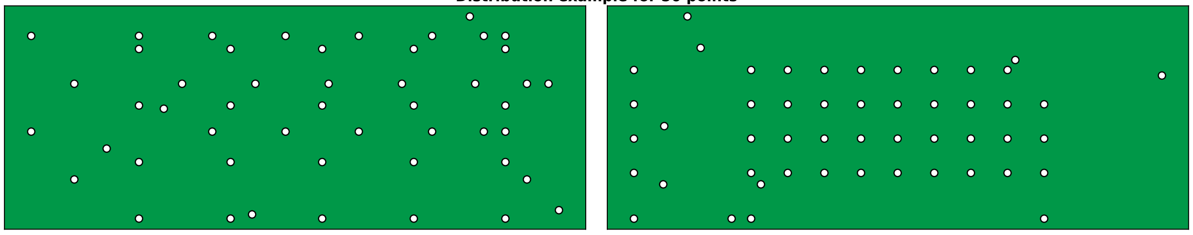
Distribution example for 30 points



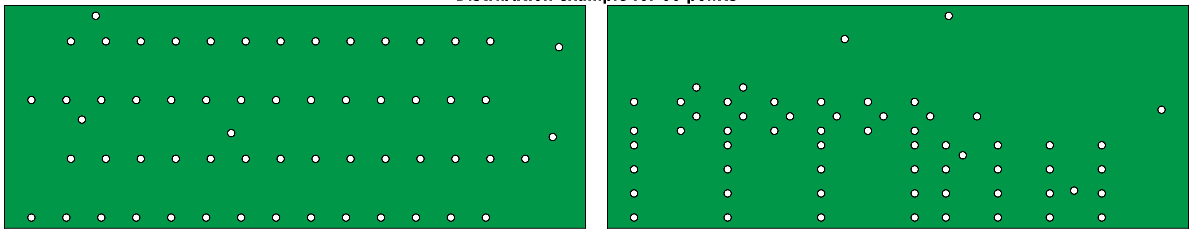
Distribution example for 40 points

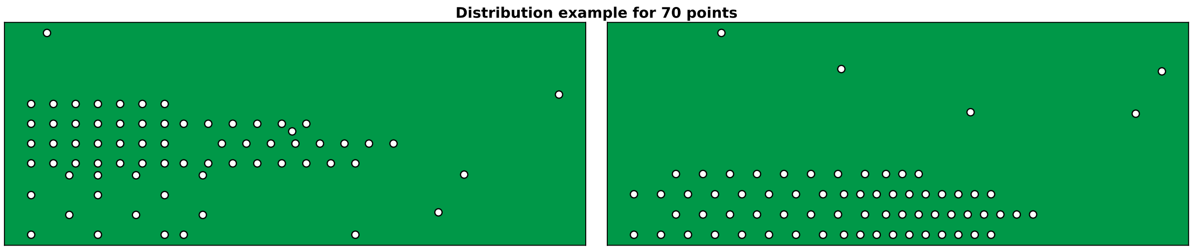


Distribution example for 50 points



Distribution example for 60 points





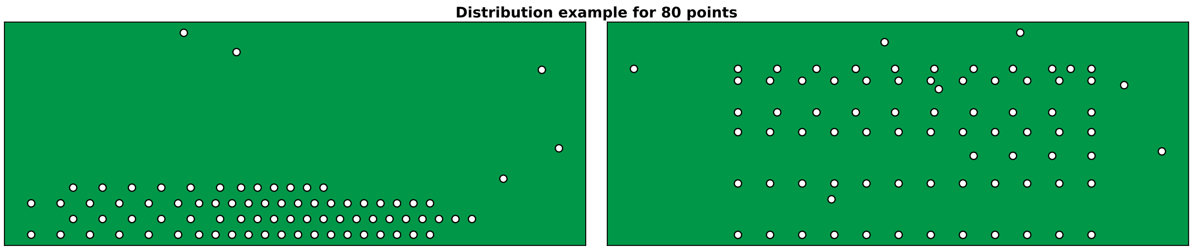


Figure 1: Sample representation of random point distributions.

3 Exact model

3.1 Implementation

The model presented in Section 1 was implemented in OPL, specifically in the file `/opl-model/hw1.mod`. It was then called using `oplrun` from command line via a python script which was iteratively passing the various time limits and the point distributions as arguments to the shell command. An example of the code to solve the optimization problem with the first instance of 40 points and a time limit of 0.1s is the following:

```
$ /opt/ibm/ILOG/CPLEX_Studio128/opl/bin/x86-64_linux/oplrun -v
  -D timeLimit=0.1 /opl-model/hw1.mod /points/40/1.dat
```

After a solution was found, the OPL solver output the details of the optimization process, which were saved in the format `/points/x/y_z_output.txt` where x indicates the points quantity, y the instance id and z the time limit constraint. The optimal value of the decision variable, representing the most efficient route, was saved in `*_sol.txt` files following the previous name pattern. When the time limit was too strict (0.1s, 1s or rarely 10s) the solver wasn't able to provide an admissible solution, this occurrence was tracked and recorded in the file `unsolvables.json`.
This process was done using the `002-optimize.py` file. Please note, before running the `002-optimize.py` file, the following script should be run:
`$ export LD_LIBRARY_PATH=/opt/ibm/ILOG/CPLEX_Studio128/opl/bin/x86-64_linux/`
and the absolute path to the OPL solver and model should be updated.

3.2 Result sample

After running the model on the various instances, the results had to be extracted from the various files generated by OPL.
Following is an example of the output solution for a 10 point instance:

```
y = [[0 1 0 0 0 0 0 0 0 0]
     [0 0 0 1 0 0 0 0 0 0]
     [1 0 0 0 0 0 0 0 0 0]
     [0 0 0 0 1 0 0 0 0 0]
     [0 0 0 0 0 1 0 0 0 0]
     [0 0 0 0 0 0 0 1 0 0]
     [0 0 1 0 0 0 0 0 0 0]
```

```
      [0  0  0  0  0  0  0  0  0  1]
      [0  0  0  0  0  0  1  0  0  0]
      [0  0  0  0  0  0  0  0  1  0]];
x  =  [[0  4  0  0  0  0  0  0  0  0]
      [0  0  0  3  0  0  0  0  0  0]
      [5  0  0  0  0  0  0  0  0  0]
      [0  0  0  0  2  0  0  0  0  0]
      [0  0  0  0  0  1  0  0  0  0]
      [0  0  0  0  0  0  0  10  0  0]
      [0  0  6  0  0  0  0  0  0  0]
      [0  0  0  0  0  0  0  0  0  9]
      [0  0  0  0  0  0  7  0  0  0]
      [0  0  0  0  0  0  0  0  8  0]];
```

As mentioned before, the decision variable x_{ij} indicates the amount of flow moving from node i to node j. Considering the above example, there are 10 units of flow available (starting from position [6,8]) and each node should receive only one unit, so the overall solution can be decoded as:

1. 10 units of flow move from node 6 (row 6) to node 8 (row 8)
2. 9 units of flow move from node 8 (row 8) to node 10 (row 10)
3. 8 units of flow move from node 10 (row 10) to node 9 (row 9)
4. ...
5. 1 unit of flow moves from node 5 (row 5) to node 6 (row 6)

A visual representation of the above solution is shown in Fig.2

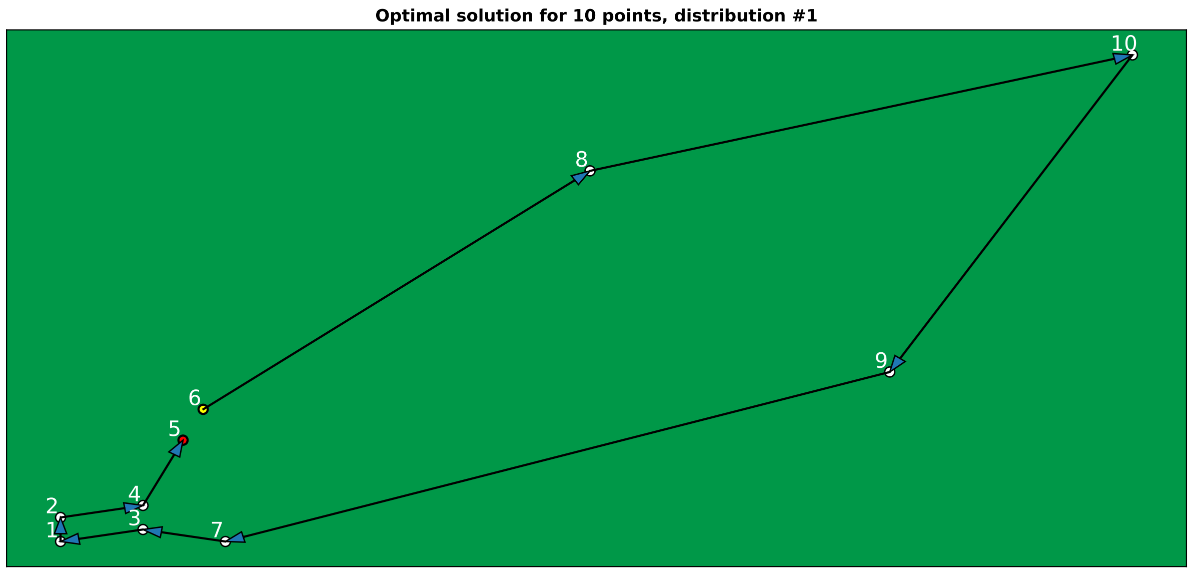


Figure 2: Optimal solution for a 10 point distribution. The last closing arc is omitted for ease of visualization. Yellow dot representing the starting point, the red one representing the last one before closing the cycle. The white numbers are the point ids.

3.3 Result extraction

The files containing the solutions for all the point distributions generated by OPL were thus parsed using the python script `003-extract-results.py` which saved the following information to an external `results.json` file:

- Running time in ms
- Objective function value
- The delta from the optimal solution

The distance from the optimal solution was computed using the following formula:

$$\Delta_{opt} = \left(\frac{x}{x^*} - 1\right) \cdot 100$$

Where x^* represents the optimal solution, obtained without any time constraint on the model, while x represents the feasible incumbent solution at the time the script terminated.

4 Heuristic approach - Genetic algorithm

Part of the goal of the project was to compare the exact optimization approach presented in Section 1 with a Heuristic approach in order to evaluate the respective strengths and weaknesses in different scenarios. A genetic algorithm was chosen for this purpose, the reason being its flexibility, the wide applicability of such approach and ease of implementation.

4.1 Model implementation

The definition and implementation of a genetic algorithm follows different steps:

1. Definition of a solution representation
2. Creation of the initial population
3. Selection of the fittest individuals as candidates for the next generation of offsprings
4. Crossover: combining the individuals
5. Mutation: introducing random mutations to some individuals
6. Replacing the previous generation with the new offsprings

The above loop is repeated from step 3 until a convergence criterion is met.
These paradigms are peculiar because, although their implementation is pretty straightforward, they require careful choice of the parameters and the definition of the techniques used in the various implementation steps. The following part will describe in more details the choices that were taken in order to implement such model.
In the context of genetic algorithms each solution is called *chromosome* and each element of the solution is a *gene*. The chromosome will represent a sequence of genes, which are the id of the nodes/points, in the order they need to be visited. So a sequence representation in the form of `[1,2,3,4]` indicates a path of 4 nodes that has to be visited in the order 1,2,3,4.
Each individual of the initial population is created by picking a random sample of the available nodes without replacement, with size coherent with the instance the model is solving. Possible random chromosomes for an instance of 4 genes/nodes are: `[[1,4,2,3], [4,3,1,2], [2,1,4,3], ...]`. The number of chromosomes generated will determine the number of individuals of the population, whose number will not vary throughout the optimization process. The main goal of this initial phase is generating diversity among the population: the best features are not yet known so it is important to generate a number of diversified individuals that cover as much as possible the solution space.
During the next phase the best individuals are extracted from the population using a *tournament* approach: a random subset, 5% the size of the population, is selected and the individual with the best fit is chosen. This process is repeated until a number of individuals equal to the size of the population is chosen, thus creating a selection of individuals for the next generation of offsprings. During this selection phase the goal is *intensification*, ie. to select the best individuals in order to combine and intensify their features. It is however important to also keep a level of diversification in order to avoid premature convergence to a local optimum, so worse individuals will be selected as well, with lower probability.
During the *Crossover* phase, the individuals are combined using order crossover with *2-cut-points*. Following this approach, two parents will generate two offsprings. Crossover between two subsequent individuals is performed with a specified probability (`cxpb=0.64`) between individual `i-1` and individual `i`. To avoid genetic drift, random mutation via 2-opt substring reversal is applied to individuals with a specific probability `mutpb=0.33`. The aforementioned approaches were chosen because they guarantee solution feasibility. It is important to note that this phase's goal is to foster genetic diversity rather than population diversity.
After mutation is completed, the new generation of offsprings fully replaces the old population and phases from 3 to 6 are repeated until convergence is met. Details on the convergence criterion are presented in the following section.

The best individual of each generation is stored and the best individual ever met in any generation represents the best solution found by the algorithm.
The optimization process is run 3 times in order to mitigate the event of a getting a good/bad solution by chance and the objective function value is computed as the mean value of the 3 runs. Using the minimum value of the objective function may be more reasonable in a production environment, however the mean seems more appropriate for evaluating the performance of the algorithm.

4.2 Additional features

One of the benefits of heuristic approaches is the potentiality to provide a feasible solution in a shorter time compared to exact approaches. It was thus important to tune the convergence criterion appropriately. The default behavior of the `deap` library is to stop when the maximum number of generations is met, but better strategies can be explored.
It is frequent to obtain substantial gains in terms of fitness during the first generations and then plateau or improve very slowly in later generations. To avoid investing excessive time in pursuing minimal gains, *early stopping* was implemented: subsequent generations were stopped if the population mean fitness in the last 16 generations didn't show an improvement of at least 0.0001 with respect to the best mean fitness value of all the previous generations.
This represented the main stopping criterion, as well as specific time limit constraints or maximum number of generations reached.

4.3 Model parameter calibration

One of the main disadvantages of genetic algorithms is the numerous amount of parameters that have to be calibrated in order to optimize its performances. This process should be done by the algorithm designer, and the choice of the parameters should be generalized to the diverse set of problems and instances the end user will apply it to.
The library `hyperopt` was used for this purpose, which applies a Bayesian approach to efficiently explore the parameter space and tries to focus in a subset of parameters which gives the best values in term of minimization of a user-defined loss function.
The loss function to minimize was defined as:

$$L(\Delta_{opt}, t_{opt}, t) = 1.2\Delta_{opt} + \frac{t}{t_{opt}}$$

where Δ_{opt} indicates the relative distance from the optimal solution obtained with OPL, t_{opt} indicates the time to find the optimal solution with OPL and t indicates the time to compute the best solution with the genetic algorithm using the current combination of parameters. The above loss function computes a weighted combination of the error and the computation time, giving more importance to a lower error, than a lower computation time.
The following parameters sets were explored:

- Population size: `[100, 150, 200, 250, 300, 350, 400, 450, 500]`
- Crossover probability: `[0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9]`
- Mutation probability: `[0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6]`
- Number of generations: `[50, 100, 150, 200, 250, 300, 350, 400, 450, 500]`
- Number of not improving generations before applying early stopping: `[5, 10, 15, 20, 25, 30]`

The parameter space exploration was run for 2500 evaluations, using random distributions of different points, from 10 to 80, in order to get a better generalization over all the dataset.
Fig.3 shows the distribution of the loss function value for all the evaluations, which definitely shows some dispersion, even if most of the points achieve values close to zero.

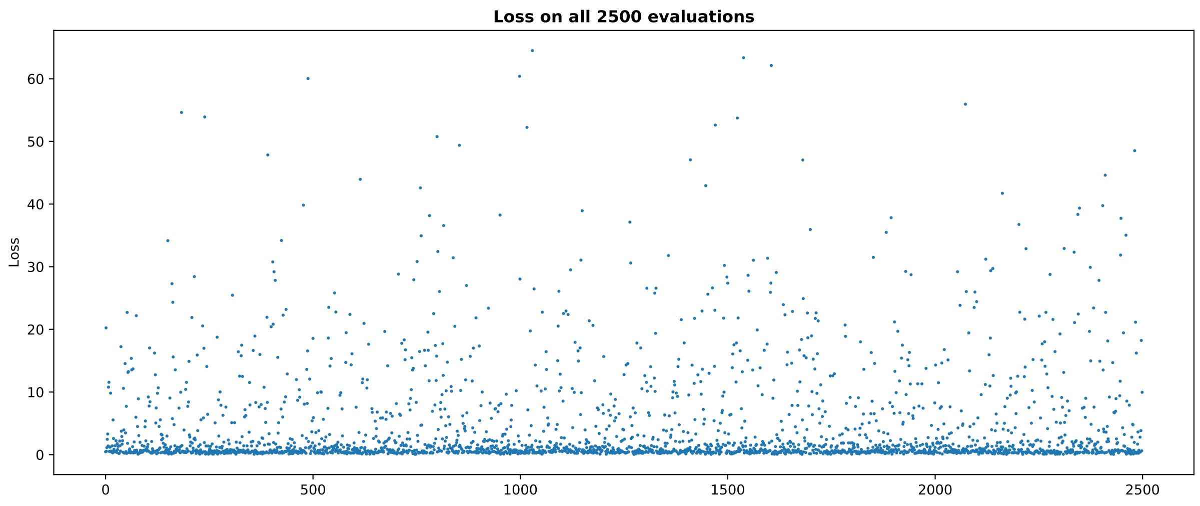


Figure 3: Distribution of loss function value for all evaluations

Exploring specific values of isolated parameters, as shown in Fig.4 doesn't show interesting results: no single parameter exhibits sensibly lower values of the loss function with respect to the others. This analysis was extended to all parameters but for conciseness only two of them are shown in this report.

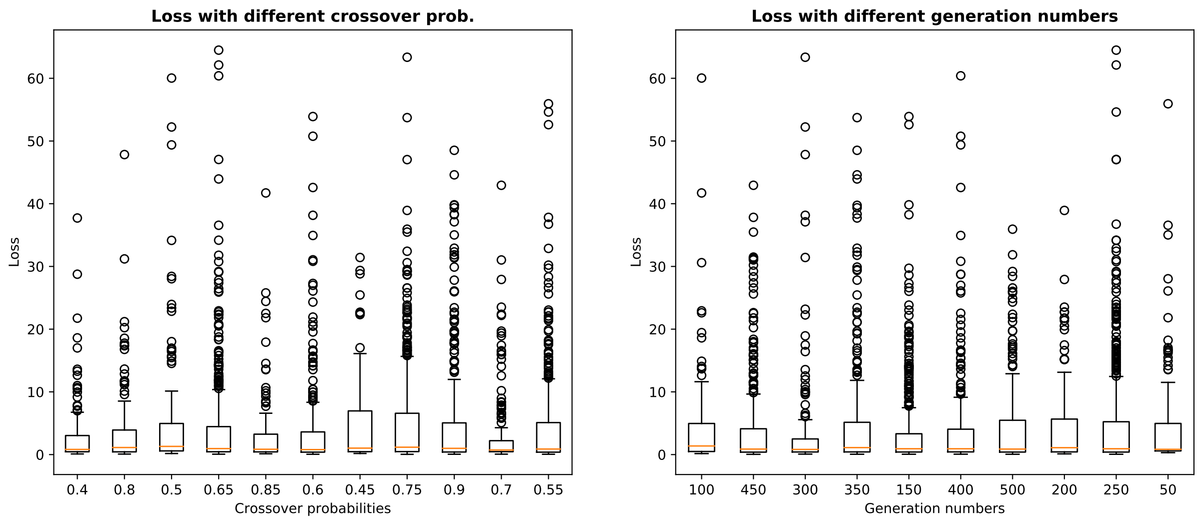


Figure 4: Loss function value distribution for two specific parameters

The library `hyperopt` returns the parameter value which achieved the lowest overall loss function value:

- Crossover P: 0.75
- Mutation P: 0.5
- Not improving limit: 10
- Number of generations: 450
- Population size: 350

But since during the 2500 evaluations the point distributions were taken at random, using the above result as optimal parameter values might lead to results which favor some distributions with respect to others. Moreover, given the intrinsic randomness of this heuristic approach, one single result won't give a reasonable generalization. To try to improve this outcome, lets analyze the 15 best values of the loss function for each point distribution, to check whether there is some consistency among all the parameter values. Fig.5 summarizes such analysis.

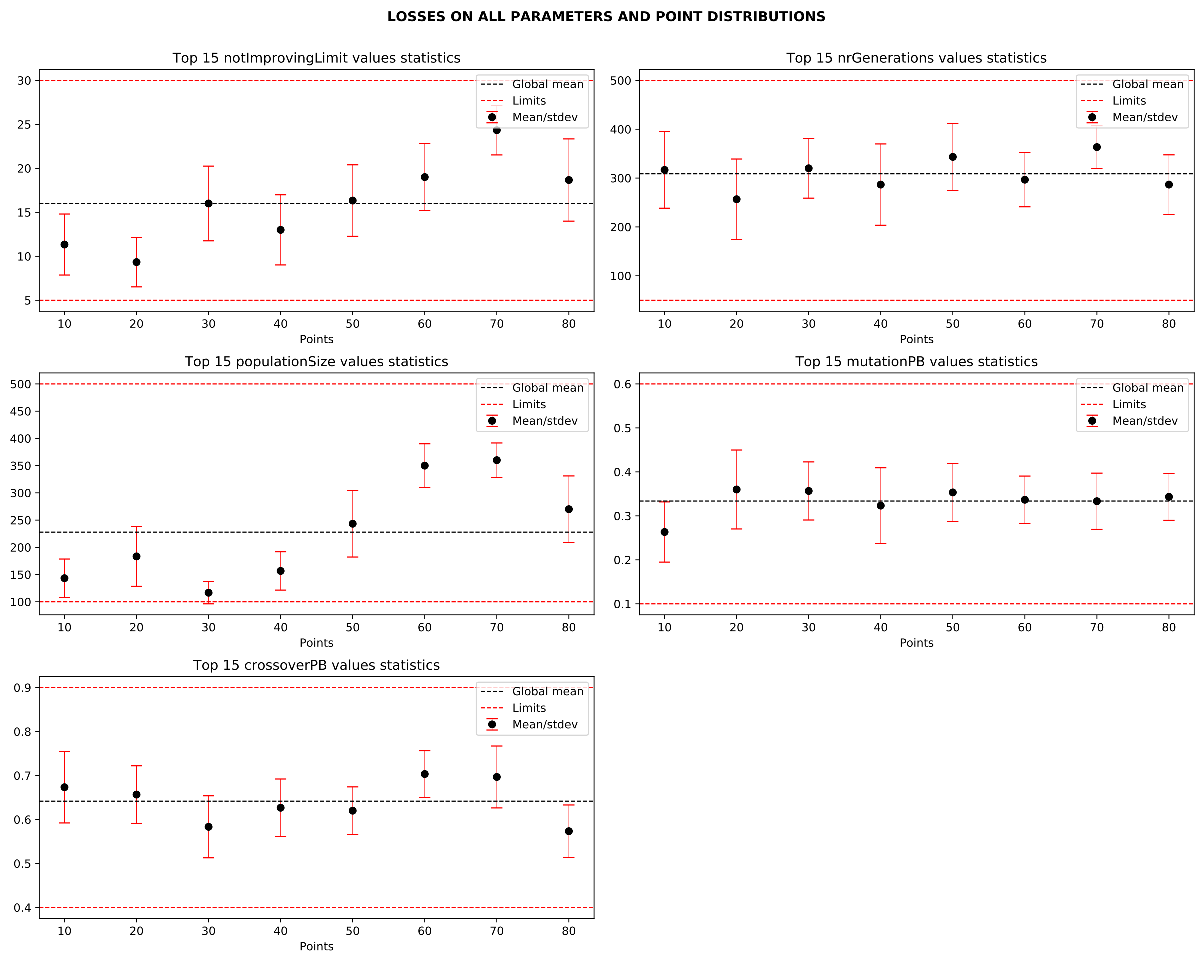


Figure 5: Parameter values for all point distributions in terms of point mean and standard deviation as well as overall mean

As it was expected, the mean values of the top 15 results for each parameter, varies significantly over different point distributions. The biggest difference is seen in the *population size* parameter as well as the *number of not improving generations*, where the mean value for distributions with small number of points differ sensibly from the ones with a higher number of points. The distributions with 80 points exhibits a peculiar pattern, which is not present in distributions with 60 or 70 points, and should be investigated further.
Since each point distribution presents different optimal values for each parameter, it is reasonable to think that defining specific values for subsets of point distributions could lead to better results, but for simplicity reasons the mean value (the black dotted line in the previous plot) was chosen as single best parameter value for all the point distributions:

- Crossover P: 0.64
- Mutation P: 0.33
- Not improving limit: 16
- Number of generations: 308
- Population size: 230

It is interesting to note that there are minor differences compared to the optimal parameter value suggested by the `hyperopt` library.
An animated plot was generated in order to have a qualitative feedback on the evolution of the solution space over the generations of a genetic algorithm implementing the above parameters. The plot, available in the file `genetic-animation.gif`. shows the superposition of all the individuals of each generation for a specific point distribution instance. Fig.6 shows a preview of the last generation frame.

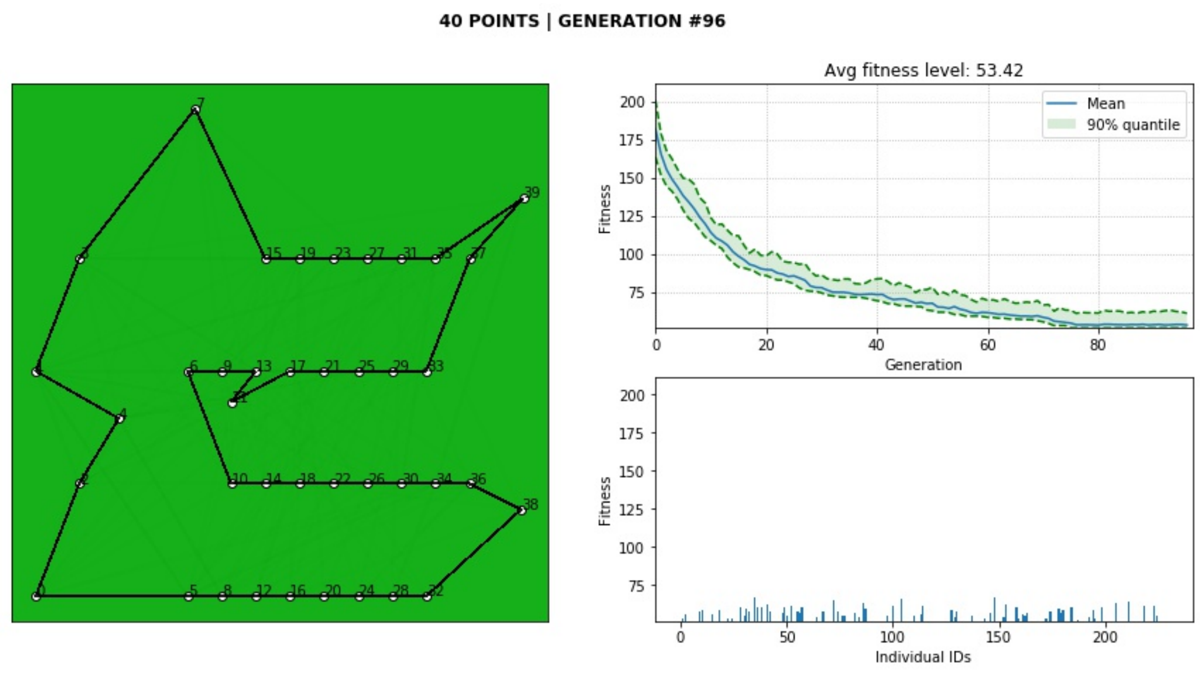


Figure 6: Last frame of the plot animation presenting the evolution of the solution space of a genetic algorithm across various generations. The animated version is available in the file `genetic-animation.gif`

5 Results

This section presents the results and the performance comparisons between the exact approach and the genetic algorithm.

5.1 Computation time comparison

One of the major benefits of genetic algorithms, besides the ease of implementation, is their ability to find a feasible solution in a short amount of time. Keeping this into consideration, one of the goals of this work was to evaluate the two approaches respectively in terms of time needed to find the optimal solution for OPL and the best possible solution with the genetic algorithm. Fig.7 summarizes the overall optimization times for both approaches. The optimization time for the genetic algorithm has to be intended as the overall time taken for all the 3 runs.

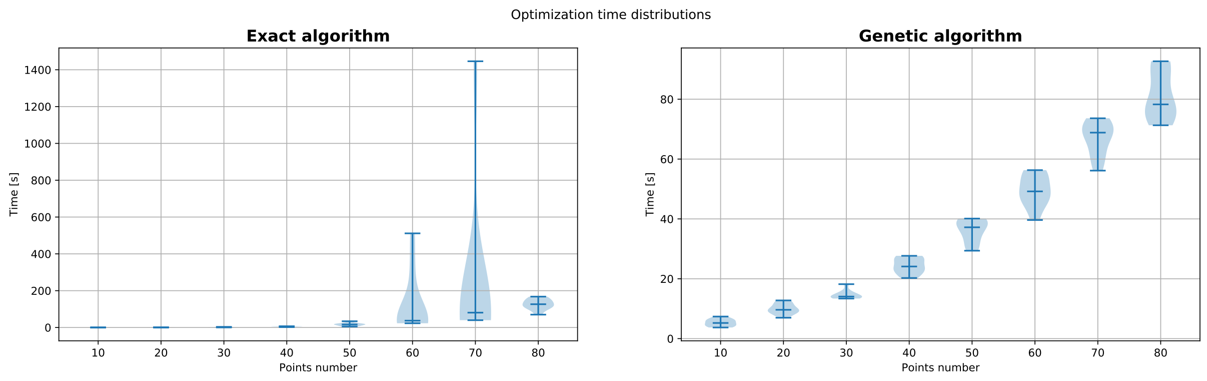


Figure 7: Optimization time for all point distributions, for both exact and genetic algorithm.

The outlier in the distribution with 70 points makes the plot difficult to compare between the two algorithms, so for ease of comparison the first plot is reproduced below with the same y-scale, excluding the aforementioned outlier.

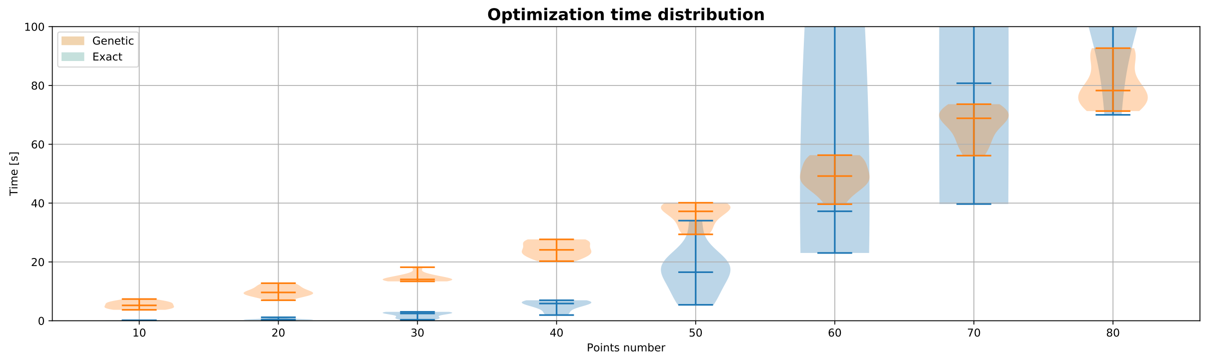


Figure 8: Overlay comparison of optimization times for exact and genetic approach in the time range [0s,100s].

As it is clear from Fig.8, the exact algorithm is able to find the optimal solution in less time than the genetic algorithm in most instances, this result is evident by looking at the respective median values. The only situations in which the genetic algorithm outperforms the exact one, in terms of computation time, are the most complicated instances with more than 70 points.
It is worth noting the sensible difference in computation time in the small instances: the OPL solver seems to be extremely fast, however it must be taken into consideration that the genetic algorithm is doing 3 runs for each instance. Moreover there might be a margin of error in the comparison of computation times in such different programming environments and the way they are computed might not be totally comparable.

5.2 Ability to provide a feasible solution

Given that the genetic algorithm starts from a population of feasible solutions, even with short time constraints on the optimization time, it will be able to provide a feasible solution. On the other hand the exact method will take some time to compute an initial solution, depending on the method used by the OPL solver. Considering these premises, it was interesting to investigate how many instances resulted unsolved by the exact method, as a function of the time limits imposed to the solver.
Fig.9 summarizes the results:

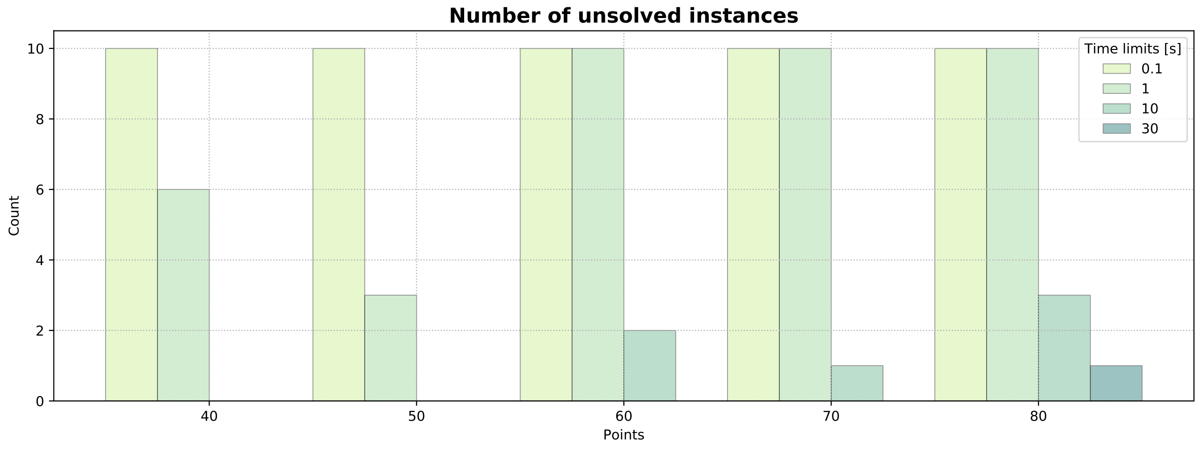


Figure 9: Number of unsolved instances by the exact algorithm as a function of the time constraint imposed to the solver.

As expected, imposing strict limit constraints, below 1 second, causes the exact algorithm not to provide any solution with most of the instances having 40 or more points. A time limit of 10 or 30 seconds seems to be critical only for particularly complicated instances having 60 or more points.

5.3 Delta from optimal solution

Beside the computation time, another crucial aspect when considering an optimization algorithm is the quality of the solution it provides in terms of distance from the optimal solution. The optimal solution is assumed as the one computed with the OPL solver using no time constraint. To review how Δ_{opt} was computed, please refer to Section 3.3.
Fig.10 shows the deltas of the objective function values from the optimal solutions for all the available point distributions and instances. It is important to recall that the objective function value for the genetic algorithm approach is computed as the mean of 3 or less runs, the following color coding is used to distinguish statistics computed on different runs:

- Red: result based on 1 run
- Yellow: result based on 2 runs
- Green: result based on 3 runs

By looking at the plots in Fig.10 there are various things to note. The deltas from the optimal solution with the exact approach shows how, as the complexity increases and the time limit constraint gets stricter, the distance from the optimal solution increases. The plots referred to the 1s and 0.1s time limit constraint are also showing less point distributions as for some of them it was not possible to retrieve any feasible solution from OPL. The OPL solver was however able to find the optimal solution for all point distributions up to 40 points, except with 1s and 0.1s time limit constraint.
The results achieved by the genetic algorithm are instead very different, almost all point distributions have a non zero delta from the optimal solution, which increases with the complexity of the instance. This again was expected, since there is no guarantee of optimality with this metaheuristic approach, but it's interesting to compare the entity of the delta as the complexity increases and the time limit constraint gets stricter. The genetic algorithm achieves a better result for 80 point distributions in all the time constrained experiments, this is also true for the 70 point distribution with time constraints below 30s. On more extreme scenarios, like the 1s time limit constraints and distributions of over 40 points, the differences are very much in favor of the genetic algorithm, with deltas that are close to one half or one third the ones of the exact approach. Lets also keep in mind that most of the results achieved on strict time limit constraints are the outcome of only one run of the genetic algorithm and are thus not very reliable.
One last thing to note is that the increase in the delta as a function of the complexity is much steeper in the exact algorithm than the genetic one, so in some specific scenarios the latter approach might scale better.

5.4 Reliability of the genetic algorithm results

The previous section introduced an interesting aspect regarding the reliability of the results obtained using the genetic algorithm: the number of runs it can perform under a strict time constraint. As it was stated before, the genetic algorithm is always capable of providing a feasible solution (if the initial population is composed by feasible individuals and crossover/mutation guarantee feasibility) but its reliability is related to the number of runs it is computed on. In fact a good objective function value calculated on a single run can be just the outcome of a random fortunate event.
With the above intuition on the concept of *reliability*, this statistic was computed on all the optimization results obtained with the genetic algorithm which are summarized in the heatmap on Fig.11 which shows the mean number of runs on all the instances and time limit constraints for each point distribution.

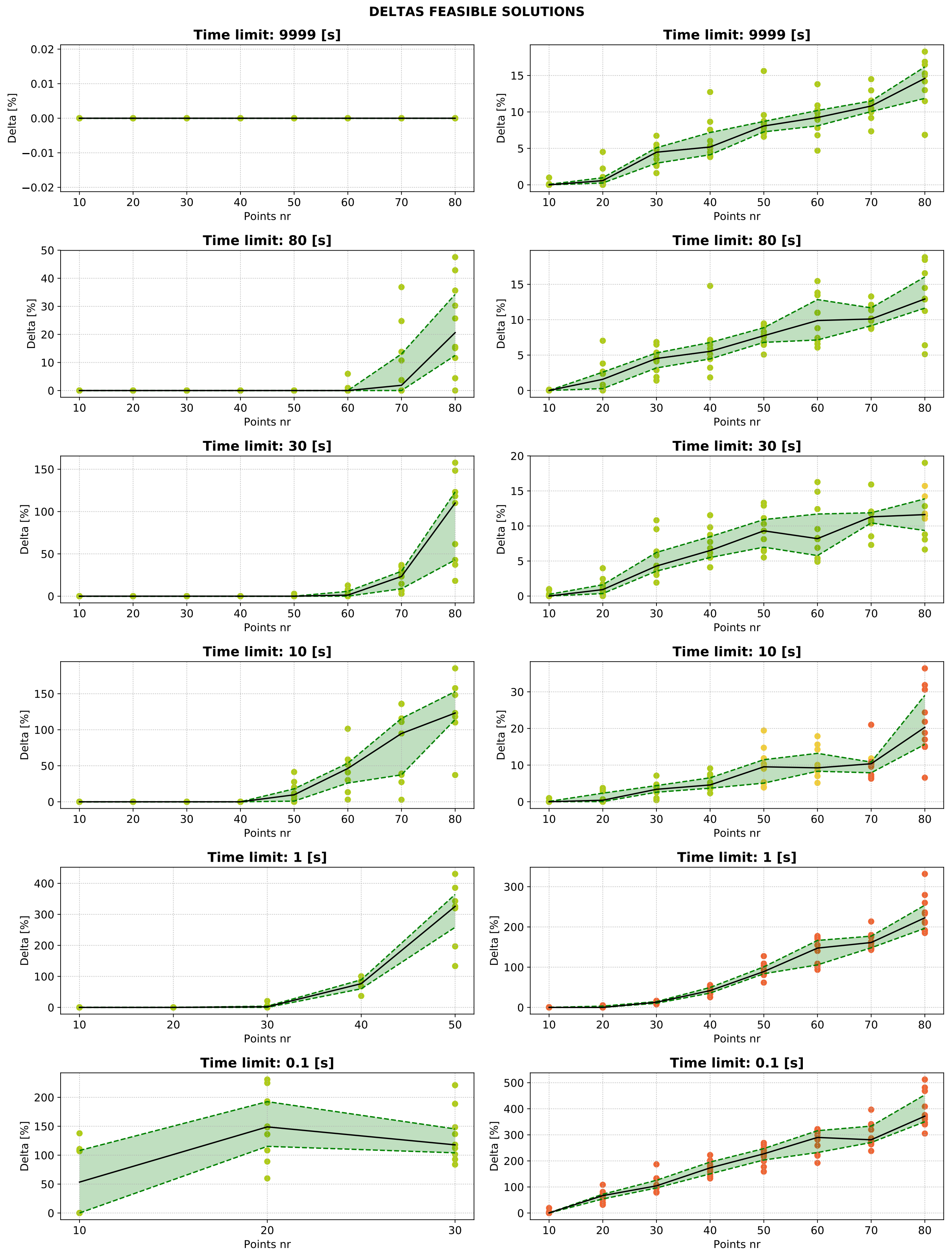


Figure 10: Deltas from optimal solutions of all the available instances on all the point distributions. The black line indicates the median and the green area is the 50% quantile around the median. For the genetic algorithm the following color coding is used: green to indicate a value computed as mean of the objective function value of 3 runs, yellow is computed as mean of 2 runs and red is computed on 1 single run.

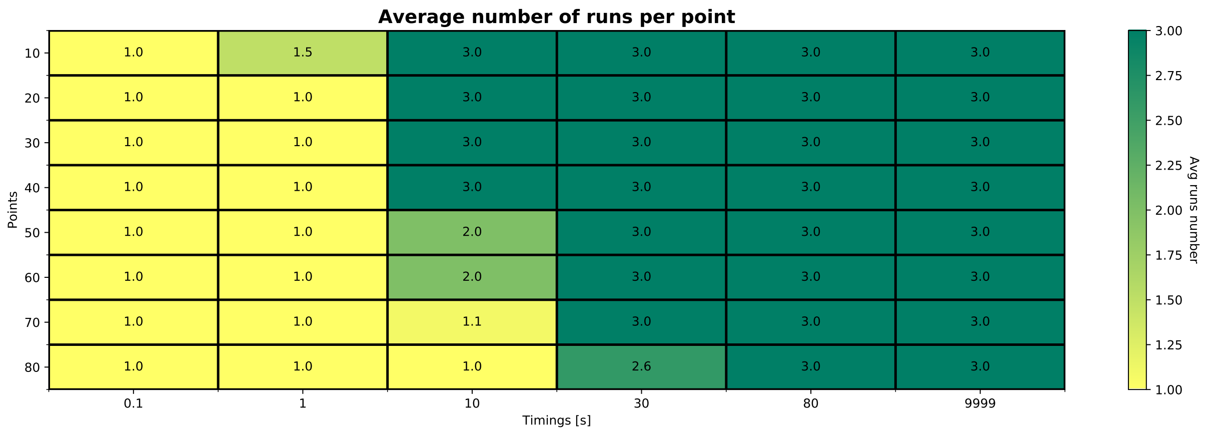
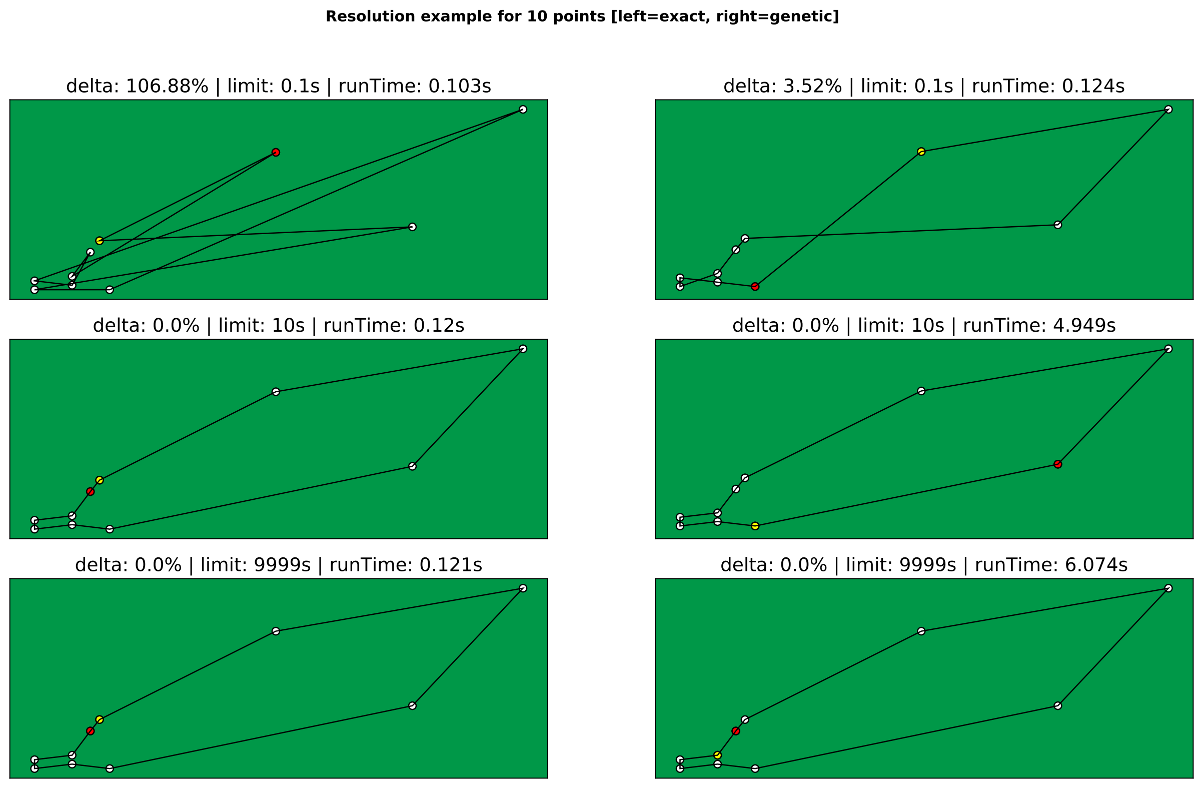


Figure 11: Average number of runs per point distribution for all the time limit constraints referred to the genetic algorithm approach.

The genetic algorithm was able to do more than 1 single run on most instances with time constraints of 10s or more, results achieved with 0.1s or 1s constraint are to be considered more unreliable since they could simply be the outcome of a random event.

5.5 Solution visual representation

As last element it was curious to have a visual representation of the solutions proposed by the two algorithms for different point distributions and different time constraints. The plots in Fig.12 give a qualitative view on this regard by sampling a limited number of the stored solutions and presenting a side by side comparison between the two approaches.



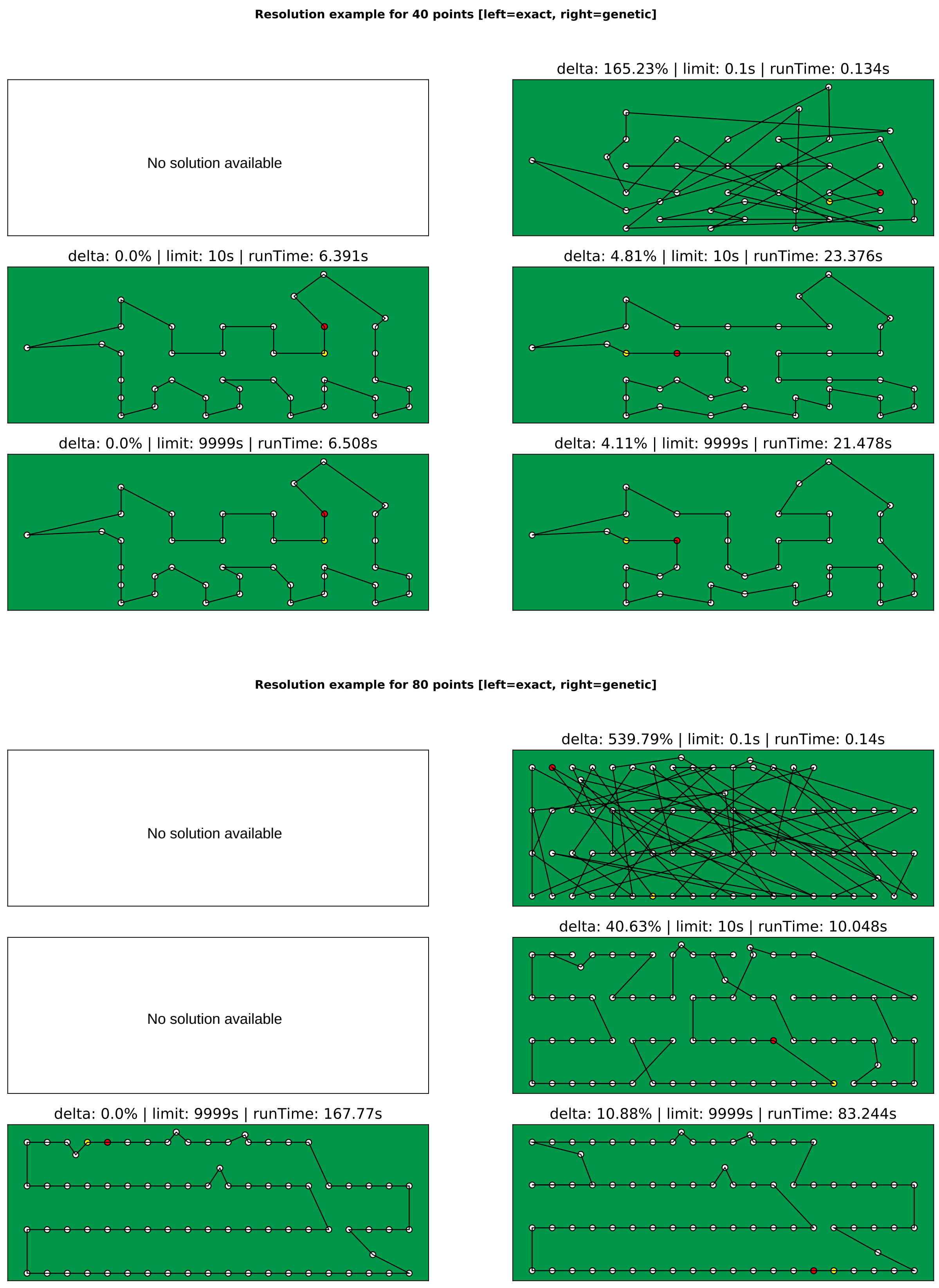

Resolution example for 40 points [left=exact, right=genetic]
No solution available
delta: 165.23% | limit: 0.1s | runTime: 0.134s
delta: 0.0% | limit: 10s | runTime: 6.391s
delta: 4.81% | limit: 10s | runTime: 23.376s
delta: 0.0% | limit: 9999s | runTime: 6.508s
delta: 4.11% | limit: 9999s | runTime: 21.478s
Resolution example for 80 points [left=exact, right=genetic]
No solution available
delta: 539.79% | limit: 0.1s | runTime: 0.14s
No solution available
delta: 40.63% | limit: 10s | runTime: 10.048s
delta: 0.0% | limit: 9999s | runTime: 167.77s
delta: 10.88% | limit: 9999s | runTime: 83.244s

Figure 12: Side by side comparison of the two approaches solutions on different point distributions and time limit constraints.