University of Padua

Department of Mathematics "Tullio Levi-Civita"

Master's Degree in Computer Science

A GENETIC ALGORITHM FOR THE TRAVELLING SALESMAN PROBLEM

Course of Methods and Models for Combinatorial Optimization

A.Y. 2019/2020

MATTEO RIZZO - 1206694

Table of contents

Abstract

The subject of the present report is the development of a Genetic Algorithm to solve a Travelling Salesman Problem (TSP) in a real-world scenario. The performances of the here proposed algorithm are compared to those of an exact method implemented via CPLEX and to those of a Tabu Search heuristic. While the exact method proves to be the obvious choice for small instances of the problem, the trends of the values of the objective function and the times of execution show, as expected, that the Genetic Algorithm is to be prefered for bigger instances.

Introduction

Installation

The code that has been produced for realizing this project is structured as described in *Fig 1* (note that many non-relevant files have been omitted).

```
data_analysis
│
├── classes
├── data
├── main__.py
└── plots

solver (project name: Labs-TSP)
│
├── classes
│   ├── solvers
│   │   ├── include
│   │   │   ├── CPLEXsolver.h
│   │   │   ├── GeneticAlgorithmSolver.h
│   │   │   ├── TabuSearchSolver.h
│   │   │   └── TSPsolver.h
│   │   └── src
│   │       ├── CPLEXsolver.cpp
│   │       ├── GeneticAlgorithmSolver.cpp
│   │       └── TabuSearchSolver.cpp
│   └── utility
│       ├── include
│       │   ├── ProblemInstanceGenerator.h
│       │   ├── TSPsolution.h
│       │   └── TSPsolverScript.h
│       └── src
│           ├── ProblemInstanceGenerator.cpp
│           ├── TSPsolution.cpp
│           └── TSPsolverScript.cpp
├── cmake
│   └── FindCplex.cmake
├── config
│   └── params.json
├── data
├── libs
│   └── rapidjson
└── main.cpp
```

Fig 1. Structure of the project.

Note that, while being just a container for the scripts related to the analysis of the data (i.e. metrics calculation and plotting) and a separate Python 3.7 project, it is important to preserve the structure of the `data_analysis` folder. The reason behind this is that the main software contained in the `solver` folder saves some output files into the `data_analysis/data` directories (so that they can be subsequently analyzed using Python instead of C++). That said, the technical details of the `data_analysis` part are not the main interest of this report and will not be further discussed.

The dependencies of the `solver` project are:

- **IBM ILOG CPLEX Optimization Studio 12.8.0**: since the project is built using Cmake, the script `cmake/FindCplex.cmake` takes care of finding a CPLEX installation within the user's machine;
- **Rapidjson**: since it is possible to configure the execution of the software via an ad hoc JSON file, the Rapidjson library was used to parse the configuration file. The library is shipped together with the rest of the code and its operation should not require any further actions. Yet, make sure that the current working directory is set to `solver`, otherwise Rapidjson will not find the configuration file. You can find more information about Rapidjson at rapidjson.org.

The specifics of the machine the project has been tested on are reported in *Table 1*.

Specific	Value
OS	Ubuntu 18.04.3 LTS 64-bit
Memory	15.5 GiB
Processor	Intel Core i7-6500U CPU @ 2.50GHz x 4

Table 1. Specifics of the machine the project has been tested on.

Configuration

The execution of the software can be configured by changing the desired parameters within the `config/params.json` file. The configuration file has the structure described in *Table 2*. An example of a working configuration that runs CPLEX on *tsp12.dat* is shown in *Fig 2.*

Parameter	Type	Description
`comparison`	`String` ∈ {"`trial`", "`size`", "`none`"}	The type of comparison among solvers to be performed: • "`trial`": executes each solver `num_trials` times on the same input instance as defined by the parameters in `size` (that in this case must feature a single size value); • "`size`": executes each solver `num_trials` times on increasing sizes of the problem instance as defined by the parameters in `size`; • "`none`": no comparison is performed. The desired `solver` is executed on the input problem instance.
`solver`	`String` ∈ {"`cplex`", "`tabu`", "`genetic`"}	The solver to be used in a single solver execution against the given `problem` and for the given sizes and times (as defined by the parameters `size` and `time`).
`problem`	`String` ∈ {"`from_file`", "`random_polygons`", "`random_symmetric_distribution`"}	The type of problem to be solved: • "`from_file`": the problem defined by the cost matrix in the input file `file_name`; • "`random_polygons`": a grid of random non-intersecting polygons; • "`random_symmetric_distribution`": a grid of random points.
`file_name`	`String`	The name of the input file in DAT format. It must feature a first line with the problem dimension followed by a cost matrix (e.g. *tsp12.dat*, *tsp60.dat*)
`num_trials`	`Int`	The number of times the input problem must be solved in a comparison.
`size`	`Int` (sub params)	The range of sizes for the size comparison. The range is defined by the `start_size`, `end_size` and `step`

		parameters. Each size $s_i,\ i \in [0,\ n]$ is obtained as $s_i = s_{i-1} + step$.
`time`	`Double` (sub params)	The range of times for the size comparison. The range is defined by the `start_time`, `end_time` and `step` parameters. Each target time $t_i,\ i \in [0,\ n]$ is obtained as $t_i = t_{i-1} * step$. When the comparison is set to "`none`", the desired solver is executed for increasing sizes up to each target time.
genetic_algorithm	`Double` (sub params)	The specific parameters of the genetic algorithm (see § *"Genetic algorithm" ≫ "Calibration of the parameters"* for more information).

Table 2. Configurable parameters for the software execution.

```
{
 "comparison": "none",
 "solver": "cplex",
 "problem": "from_file",
 "filename": "tsp12.dat",
 "num_trials": 100,
 "size": {
   "start_size": 10,
   "end_size": 60,
   "step": 1
 },
 "time": {
   "start_time": 0.1,
   "end_time": 100,
   "step": 10
 },
 "genetic_algorithm": {
   "max_iter": 500,
   "tolerance": 10,
   "population_size": 500,
   "elite_percentage": 0.1,
   "mutation_percentage": 0.15,
   "mutation_rate": 0.30,
   "k": 0.1
 }
}
```

Fig 2. Example of a working configuration for the software.

Definition of the problem

The project aims at resolving instances of the Travelling Salesman Problem (TSP) in the following scenario: an industrial machinery uses a drill to make a certain number N of holes at some given coordinates on a board of size $W \times H$; it is required to find an ordered sequence of holes s that minimizes the total drilling time, taking into account that the time needed for drilling a hole is equal and constant for all the holes.

The task is therefore modelled as the solution to a TSP on a graph where:

- The nodes are the holes to be drilled on the board, and
- The edges are the movements performed by the machine to shift from the drill position of a hole to another, which have a certain cost

We assume that the graph is complete. This means that given a set of coordinates the machine can move from any drill position towards any other one.

If we assume that the N holes to be drilled are numbered from 0 to $N-1$, then a solution is encoded as a sequence $s = (0\ s_0\ 0)$ of length $N+1$, for any subsequence $s_0 = i_1 \dots i_{N-1}$ with $i \in [1, N-1]$, $i_k \neq i_j\ \forall\ k,j \in [1, N-1]$. In other words, the initial and final node of each solution is necessarily the node 0. For example, a solution for an instance of 5 nodes may be encoded as $s = (0\ 1\ 2\ 3\ 4\ 0)$, that is a tour the starts from node 0, passes through each node $i \in [1,4]$ and comes back to 0. Then the set of all possible solutions to the problem with N nodes is the set of all possible sequences that feature a permutation without repetition of the subsequence s_0 (which has $(N-1)!$ as size). Note that the selected encoding will influence directly some choices of implementation in the Genetic Algorithm, such as the ordered crossover recombination (described in § *"Genetic algorithm" ≫ "Implementation"*).

Generation of the instances

> What is described in this section is handled by the class `ProblemInstanceGenerator`.

In order to properly simulate the real-world problem of reference (i.e. an industrial machine drilling holes on boards), problem instances must be generated such that a certain logical criterion is followed in the distribution of the points in the considered space. This situation

has been modelled by the generation of **random non-intersecting polygons** on a bounded Cartesian plane. Considering the space of coordinates in the first quadrant of the Cartesian plane, this is bounded by two lines defining together with the x and y axes the edges of the board to be drilled. Then, given a desired positive integer number n of holes as input, this is randomly split into k numbers $n_0, \dots, n_{k-1}$ such that $n_0 + \dots + n_{k-1} \leq n$. Each of the k numbers $n_0, \dots, n_{k-1}$ describes the number of edges of one of the polygons that will be generated according to a random angle and a random centre within the region described by the board (of course, $n_i \geq 3 \;\; \forall \; i \in [0, k-1]$). The cost matrix of the related problem is subsequently generated calculating the euclidean distance between each vertex of the polygons. Some examples of the generated polygons are shown in *Fig 2-3-4*.

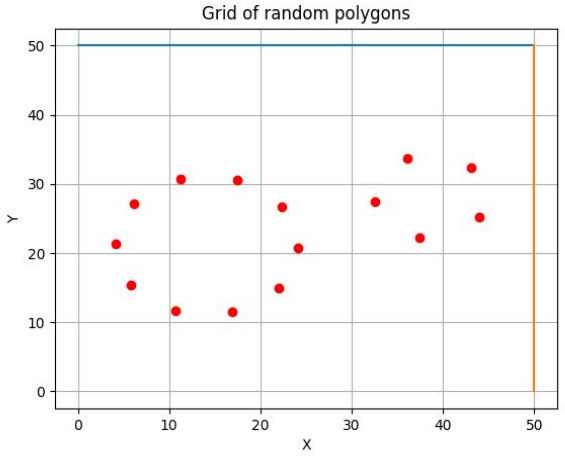

Fig 3. Example of 2 random polygons generations for 15 holes.

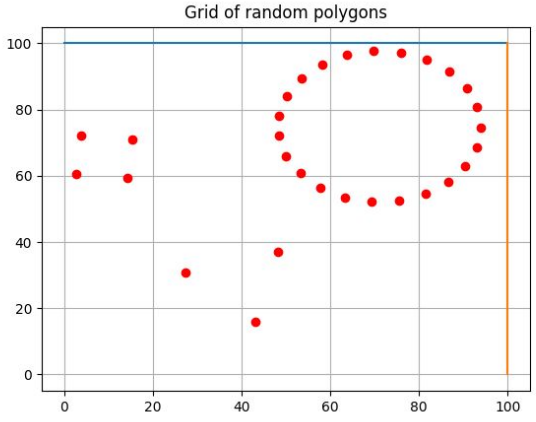

Fig 4. Example of 3 random polygon generations for 30 holes.

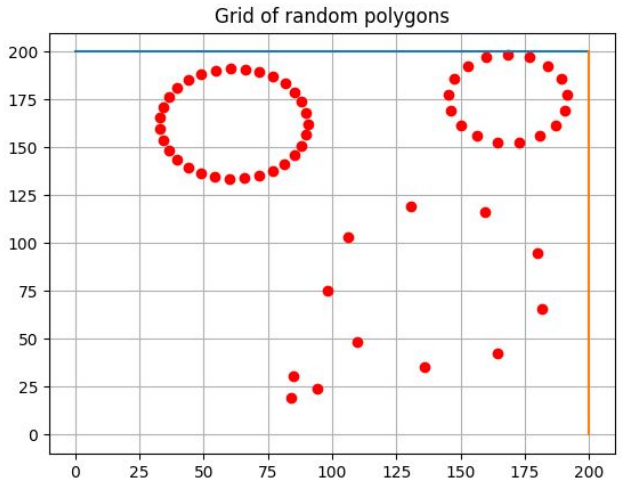

Fig 5. Example of 4 random polygon generations for 60 holes.

Note that, In addition to the non-intersecting polygons, it is also possible to generate a cost matrix as a **random distribution** of positive integer values. This kind of instance of the problem is used as further metrics in the testing of the genetic algorithm. The software also allows loading a problem instance **from a file** in DAT format (e.g. *tsp12.dat*, *tsp60.dat*). These files must contain a first row with the size of the problem followed by a cost matrix.

Optimization methods

The heuristic method that has been developed and analyzed to solve the TSP is a genetic algorithm, whose implementation is detailed further on in this report. In order to properly compare the performance of the algorithm, it has been tested against an exact method (via CPLEX) and a Tabu Search heuristic (the one proposed in the laboratory “Neighbourhood search for the symmetric TSP”). Note that the Tabu Search heuristic is not the main topic of this report and therefore it is not discussed. This heuristic has only been wrapped by a proper class to fit the structure of the project and been used as execution time and solution accuracy comparison for the genetic algorithm.

Exact method via CPLEX

What is described in this section is handled by the class `CPLEXsolver`.

The software features an exact solver that basically wraps the calls to the methods offered by CPLEX into a custom class. This class sets up the problem initializing variables and constraints according to the mathematical formulation provided in the first assignment and then optimizes it. More specifically, x and y variables, which represent the quantity of "flow" that crosses each connection between nodes and the effective usage of each edge in the solution respectively, are first added to the CPLEX environment. Each time a variable is inserted, the corresponding entry in a map is updated with the value of the index of the edge that has been inserted. This ensures the possibility of obtaining each index of both kinds of variables by simply accessing the element corresponding to the two nodes involved. Subsequently, the model constraints related to the variables are initialized:

1. The flow leaving the initial node must be exactly equal to the total number of nodes;
2. The difference in flow between the outcoming and incoming edges of each other node must be equal to one;
3. For each node, only one incoming and one outcoming edge must be used;
4. If an edge is crossed by the flow then the corresponding variable y must indicate that that edge is used.

Genetic algorithm

Implementation

What is described in this section is handled by the class `GeneticAlgorithmSolver`.

First of all, the genetic algorithm randomly initializes a population of the specified size. The initialization is performed via random swaps on the sequence $0\ 1\ ...\ N\ 0$, where N is the size of the problem instance. Then it calculates the fitness of each individual (i.e. a feasible solution) as the reciprocal of the sum of the distances between subsequent chromosomes (i.e. nodes) in its sequence, that is the reciprocal of the value of the objective function for the considered solution. The fitness values are kept sorted in descending order.

The procedure then iterates the following phases up to the fixed number of iterations or respecting the fixed tolerance:

1. **Breeding**. Individuals are recombined and the population is updated. The fittest individuals of the previous generation are preserved via elitism. The parents of a child solution are decided according to the K-tournament selection criteria, with K being parametrized on the size of the problem and regulated by the parameter `k` in the JSON configuration file. With this method, K random individuals are selected from the population and only the fittest one is allowed to recombine. The recombination happens via a 2-cut-point ordered crossover, which assures the preservation of the feasibility of the solution. The approximated complexity of the phase is $O(population\_size * (2K + 2N))$;

2. **Education**. The whole population is educated via a local search using a 2-Opt swap procedure. The approximated complexity of the phase is $O(population\_size * N^3)$. Note that this complexity is very high and is the dominant complexity. On the other hand, this phase ensures to find a good solution in a much lower number of iterations;

3. **Mutation**. A certain percentage of the population (regulated by a proper parameter) is swap mutated. This is done to contrast genetic drift and to avoid local minima. Precisely, each chromosome of the sequence of a mutated individual is randomly

swapped according to the specified mutation rate. Note that the number of the mutated individuals will be at most the one defined by the percentage of the population specified by the parameter, but could also be inferior since some individuals may mutate more than once. The individuals of the population to be mutated are picked at random from the set of the population itself minus the elite. The approximated complexity of the phase is $O(population\_size * N)$;

4. **Update of the fitness scores**. The fitness scores are updated and sorted in descending order as previously described. The approximated complexity of the phase is $O(population\_size * N)$.

Calibration of the parameters

Method

The here proposed implementation of the Genetic Algorithm features many parameters, which is one of its main cons. These parameters must be properly calibrated in order to deliver the algorithm. The tunable parameters of the algorithm are reported in *Table 3*.

Parameter	Type	Description
`max_iter`	`Nat`	The maximum number of iterations (i.e. generations) that the algorithm is allowed to perform.
`tolerance`	`Nat`	The maximum number of consecutive iterations (i.e. generations) yielding the same best value of the objective function that the algorithm is allowed to perform.
`population_size`	`Nat`	The number of individuals (i.e. solutions) stored at each iteration of the algorithm.
`elite_percentage`	`Double` $\in [0, 1]$	Percentage of individuals in the population that will go straight to the next generation without being subjected to mutation.
`mutation_percentage`	`Double` $\in [0, 1]$	The maximum percentage of individuals in the population that will be subjected to mutation.
`mutation_rate`	`Double`	Probability with which the swap mutation of a gene will

	$\in [0, 1]$	occur within the mutation of an individual of the population.
`k`	`Double` $\in [0, 1]$	Percentage of the number of nodes of the current instance of the problem that will represent the size of the K-tournament selection (e.g. $k = 0.1$ on *tsp60.dat* $\Rightarrow$ 6-tournament selection). Note that the minimum value of K is set to 2.

Table 3. Parameters of the genetic algorithm.

In order to tune the parameters, 10 executions on the *tsp60.dat* problem were used for testing each interesting value. For each instance, each parameter of the problem was calibrated with respect to the initial configuration shown in *Table 4*.

Parameter	Value
`max_iter`	250
`tolerance`	5
`population_size`	500
`elite_percentage`	0.10
`mutation_percentage`	0.10
`mutation_rate`	0.05
`k`	0.10

Table 4. Initial configuration for the calibration of the parameters.

The metrics that have been calculated on the base of the multiple executions are:

- Mean Time of Execution (MTE) to solve the problem;
- Mean Objective Value (MOV) of the heuristic solution;
- Mean Error (MI) of the heuristic solution with respect to the exact method.

Each possible value of each parameter p is assigned a score calculated as:

$$Score_p = 1 / (MTE_p * ME_p + 1)$$

The aim of this score is to highlight the best trade-off between the time of execution and the accuracy of the heuristic solution. The results of the calibration are therefore reported in *Table 5-6-7-8-9-10*. Note that the exact solution to this instance of the problem is 629.800.

Details of the calibration

Tolerance

Parameter `tolerance`	MTE (s)	MOV	ME (%)	Score
1	3.513	632.709	0.462	0.381
5	5.625	631.160	0.215	0.451
10	**7.985**	**629.800**	**0.000**	**1.000**
15	10.59	630.360	0.088	0.515
20	11.92	632.649	0.452	0.156

Table 5. Calibration of the parameter "`tolerance`".

Population

Parameter `population_size`	MTE (s)	MOV	ME (%)	Score
100	1.033	643.680	2.203	0.305
250	2.314	633.470	0.582	0.425
500	**5.904**	**630.080**	**0.044**	**0.792**
750	9.136	630.020	0.034	0.758
1000	10.201	630.420	0.098	0.498

Table 6. Calibration of the parameter "`population_size`".

Parameter `elite_percentage`	MTE (s)	MOV	ME (%)	Score
0.10	**5.887**	**630.020**	**0.034**	**0.829**

0.15	5.944	632.450	0.420	0.285
0.20	5.685	630.460	0.104	0.626
0.25	5.349	630.490	0.109	0.630
0.30	5.976	630.709	0.144	0.536
0.35	6.223	630.859	0.168	0.488
0.40	5.810	630.170	0.058	0.745

Table 7. Calibration of the parameter "`elite_percentage`".

Mutation

Parameter `mutation_percentage`	MTE (s)	MOV	ME (%)	Score
0.01	5.599	630.760	0.152	0.539
0.05	5.417	630.180	0.060	0.753
0.10	5.345	631.060	0.200	0.483
0.15	**5.575**	**629.900**	**0.015**	**0.918**
0.20	5.197	633.190	0.538	0.263
0.25	6.420	630.220	0.066	0.700

Table 8. Calibration of the parameter "`mutation_percentage`".

Parameter `mutation_rate`	MTE (s)	MOV	ME (%)	Score
0.01	6.192	630.640	0.133	0.547
0.05	5.506	631.220	0.225	0.446
0.10	6.226	630.140	0.053	0.748
0.15	5.156	630.780	0.155	0.554
0.20	5.140	630.439	0.101	0.656
0.25	5.638	630.060	0.041	0.811
0.30	**5.419**	**629.900**	**0.015**	**0.920**

Table 9. Calibration of the parameter "`mutation_rate`".

K-tournament selection

Parameter `k`	MTE (s)	MOV	ME (%)	Score
0.1	**6.330**	**630.640**	**0.133**	**0.542**
0.2	6.703	632.350	0.404	0.269
0.3	7.039	634.100	0.682	0.172
0.4	7.653	634.810	0.795	0.141
0.5	7.763	632.650	0.452	0.221

Table 10. Calibration of the parameter "`k`".

Results

After analysing the calibration of the parameters, the best values proved to be those reported in *Table 11*. The parameter `max_iter` has not been actually calibrated since it is substantially dominated by the parameter `tolerance`. It was set to an arbitrarily high value based on the fact that multiple experiments have proved that the number of iteration performed by the algorithm hardly ever surpasses 250.

Parameter	Value
`max_iter`	500
`tolerance`	10
`population_size`	500
`elite_percentage`	0.10
`mutation_percentage`	0.15
`mutation_rate`	0.30
`k`	0.10

Table 11. Calibrated configuration of the parameters of the genetic algorithm.

Tests

Reference instances

The genetic algorithm has been tested against the two reference instances *tsp12.dat* and *tsp60.dat*, which can be found in the `data` folder of the project. For each instance, the problem has been solved 100 times and the mean, standard deviation, maximum and minimum of the series of the recorded values of the objective function and times of execution have been considered. These results are shown in *Table 12-13-14-15* and discussed in the following sections.

TSP12

Objective function

The comparison of the values of the objective function for the testing on *tsp12.dat* is reported in *Table 12*. For this small instance, the Genetic Algorithm performs as well as the exact method, while the Tabu Search is only slightly inaccurate (with a mean error of the 2% ca.). However, looking at *Fig 6* it is also clear that the Tabu Search shows a notable variance.

Method	Mean	Std. Dev.	Min	Max
CPLEX	66.400	0.000	66.400	66.400
Tabu Search	67.775	1.839	66.400	72.000
Genetic Algorithm	66.400	0.000	66.400	66.400

Table 12. Comparison of the values of the objective function on *tsp12.dat*.

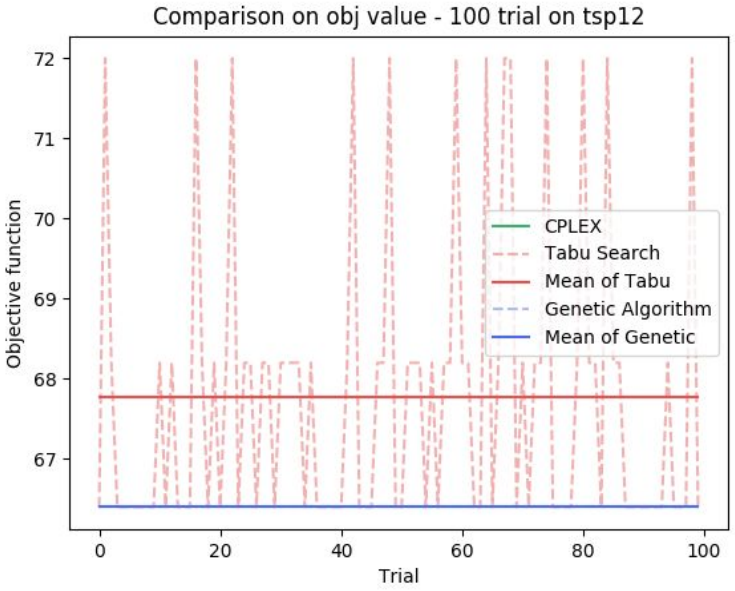

Fig 6. Values of the objective function for 100 trials against *tsp12.dat*, compared to the exact method and the Tabu Search heuristic.

Time of execution

The comparison of the times of execution for the testing on *tsp12.dat* is reported in *Table 13*. Even for this small instance, the Genetic Algorithm shows considerably high times of execution with respect to the exact method and to the Tabu Search, which instead is very fast. On the other hand, looking at *Fig 7* its time performance seems quite stable.

Method	Mean	Std. Dev.	Min	Max
CPLEX	0.055	0.012	0.049	0.160
Tabu Search	1.169e-04	2.949e-05	6.800e-05	2.290e-04
Genetic Algorithm	0.307	0.006	0.299	0.342

Table 13. Comparison of the times of execution on *tsp12.dat*.

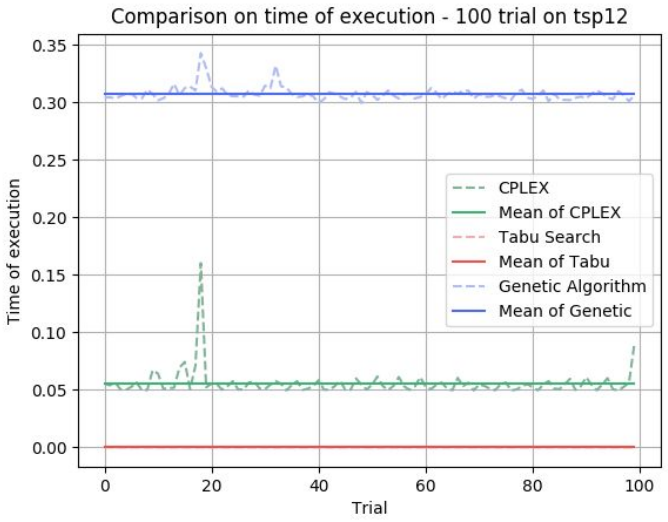

Fig 7. Times of execution for 100 trials against *tsp12.dat*, compared to the exact method and the Tabu Search heuristic.

TSP60

Objective function

The comparison of the values of the objective function for the testing on *tsp60.dat* is reported in *Table 14*. For this bigger instance, the Genetic Algorithm still performs better than the Tabu Search, while showing a reasonable variance. Tabu Search is again slightly inaccurate (with a mean error of the 3% ca.) and with a notable variance. These observations are made on the basis of *Fig 8*.

Method	Mean	Std. Dev.	Min	Max
CPLEX	629.800	0.000	629.800	629.800
Tabu Search	649.040	13.008	629.800	704.000
Genetic Algorithm	630.969	2.803	629.800	648.300

Table 14. Comparison of the values of the objective function on *tsp60.dat*.

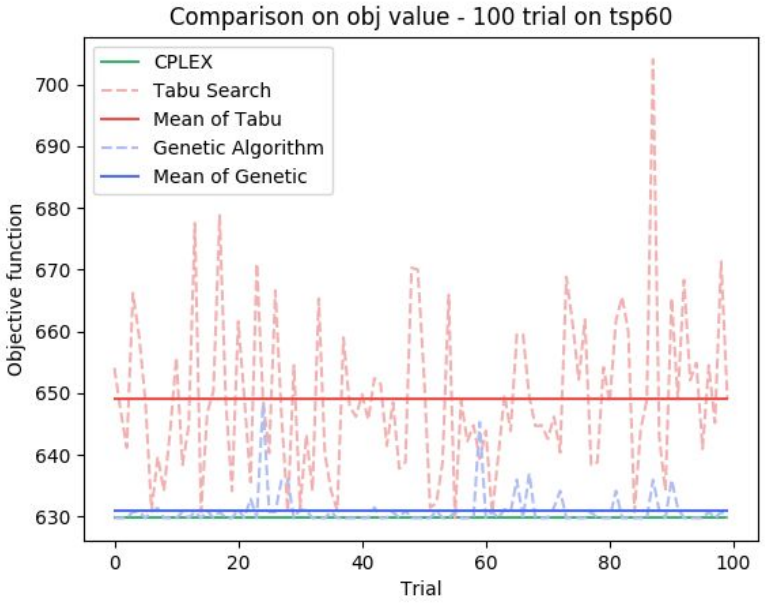

Fig 8. Values of the objective function for 100 trials against *tsp60.dat*, compared to the exact method and the Tabu Search heuristic.

Time of execution

The comparison of the times of execution for the testing on *tsp60.dat* is reported in *Table 15*. For this bigger instance, the Genetic Algorithm performs better than the exact method, but it is definitely slower than the Tabu Search while showing a reasonable variance. This is clear when looking at *Fig 9*.

Method	Mean	Std. Dev.	Min	Max
CPLEX	24.749	0.489	24.363	27.551
Tabu Search	0.001	4.593e-04	0.001	0.005
Genetic Algorithm	6.968	0.645	5.613	8.746

Table 15. Comparison of the times of execution on *tsp60.dat*.

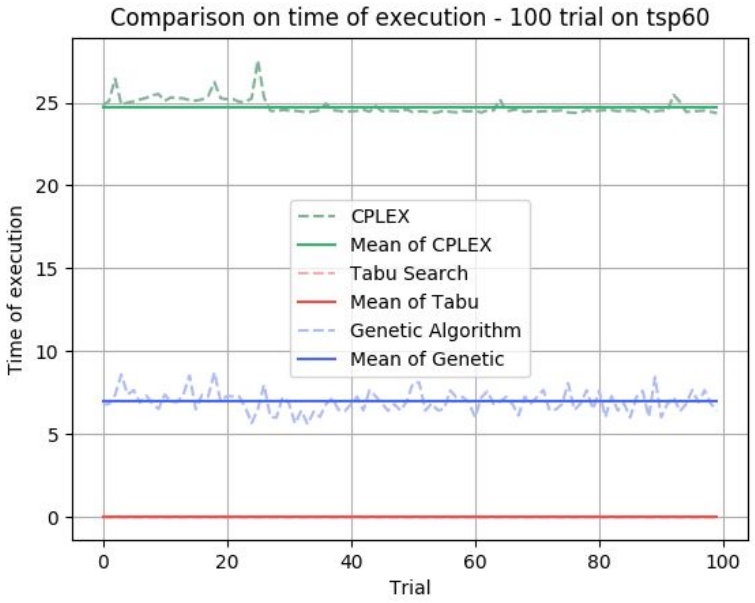

Fig 9. Times of execution for 100 trials against *tsp60.dat*, compared to the exact method and the Tabu Search heuristic.

Random polygons

In order to analyze the trend of the solutions and the times of execution, random instances of the "random polygons" problem have been generated with sizes from 5 to 70 nodes with step 5. For each size, 10 different instances of the problem have been solved. Then the mean, standard deviation, maximum and minimum of the series of the recorded values of the objective function and times of execution have been considered. These results are shown in *Table 16-17-18*.

From *Fig 10* it is clear that both the heuristic methods have proven to be able to find solutions that are very close to the global minimum for this kind of problem. On the other hand, in *Fig 11* the Genetic Algorithm shows times of execution considerably higher than those of Tabu Search. However, while the times of the Genetic Algorithm grow mostly slowly and steadily, the exact method shows quite an irregular trend and on average proves to be faster.

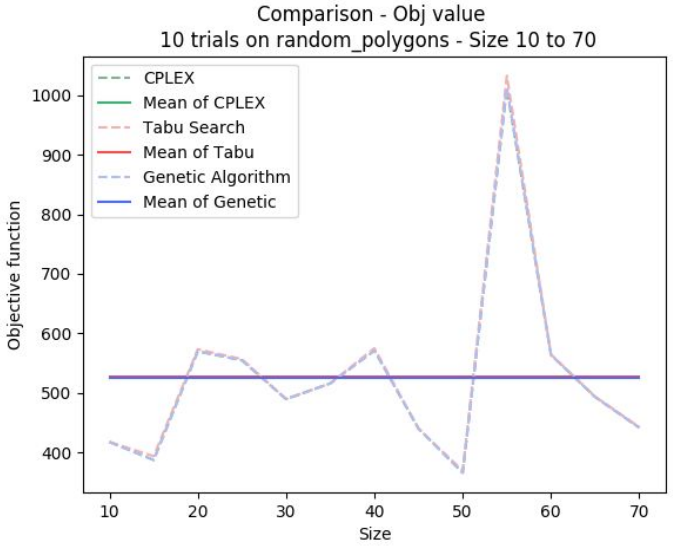

Fig 10. Values of the objective function for 10 trials against increasing size of random polygons, compared to the exact method and the Tabu Search heuristic.

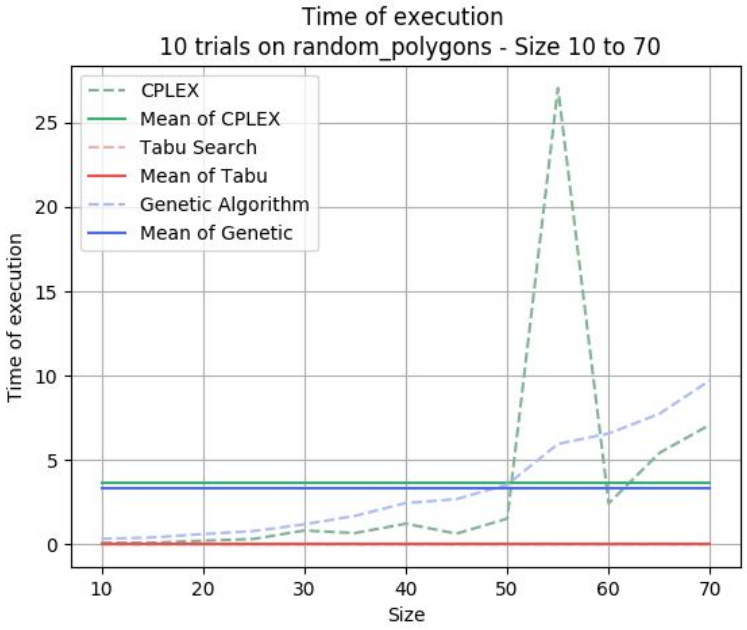

Fig 11. Times of execution for 10 trials against increasing size of random polygons, compared to the exact method and the Tabu Search heuristic.

Exact method via CPLEX

Size	Objective function	Times of execution			
		Mean	Std. Dev.	Min	Max
10	417.552	0.100	0.063	0.037	0.268
15	387.254	0.105	0.031	0.073	0.178
20	569.709	0.222	0.034	0.194	0.293
25	554.679	0.327	0.050	0.282	0.460
30	489.527	0.826	0.021	0.799	0.870
35	515.830	0.677	0.093	0.607	0.902
40	571.006	1.231	0.068	1.125	1.337
45	439.870	0.642	0.014	0.614	0.662
50	365.750	1.529	0.083	1.457	1.767
55	1010.930	27.050	0.914	25.567	28.688
60	563.655	2.436	0.081	2.373	2.658
65	493.074	5.423	0.141	5.236	5.677
70	441.383	7.104	0.877	6.021	8.897

Table 16. Values of the objective function and time of execution for CPLEX on increasing size of random polygons.

Tabu search

Size	Values of the objective function				Times of execution			
	Mean	Std. Dev.	Min	Max	Mean	Std. Dev.	Min	Max
10	417.552	0.000	417.552	417.552	1.02e-04	2.50e-05	6.80e-05	1.50e-04
15	393.677	5.532	387.254	405.637	1.29e-04	2.20e-05	1.13e-04	1.87e-04
20	573.139	5.389	569.709	588.326	6.98e-04	3.39e-04	1.77e-04	0.001
25	556.683	2.475	554.679	561.141	9.74e-04	2.18e-04	6.53e-04	0.001
30	489.527	0.000	489.527	489.527	9.91e-04	2.05e-04	7.62e-04	0.001
35	516.030	0.601	515.830	517.833	0.001	2.75e-05	0.001	0.001
40	574.927	2.535	571.006	577.591	9.41e-04	6.15e-05	9.01e-04	0.001
45	439.870	0.000	439.870	439.870	0.001	2.21e-04	0.001	0.001
50	369.331	9.745	365.750	398.554	0.001	2.49e-04	0.001	0.001

55	1032.382	7.002	1017.770	1041.046	0.001	2.69e-04	0.001	0.001
60	564.448	1.237	563.640	566.491	0.001	1.16e-04	0.001	0.001
65	493.604	0.649	493.074	494.976	0.001	2.35e-04	0.001	0.002
70	442.689	2.325	441.383	447.770	0.002	2.78e-04	0.002	0.003

Table 17. Values of the objective function for the Tabu Search on increasing size of random polygons.

Genetic algorithm

Size	Values of the objective function				Times of execution			
	Mean	Std. Dev.	Min	Max	Mean	Std. Dev.	Min	Max
10	417.552	0.000	417.552	417.552	0.329	0.006	0.320	0.342
15	387.254	0.000	387.254	387.254	0.414	0.010	0.399	0.430
20	569.709	0.000	569.709	569.709	0.617	0.050	0.533	0.691
25	554.679	0.000	554.679	554.679	0.791	0.050	0.728	0.872
30	489.527	0.000	489.527	489.527	1.195	0.073	1.065	1.268
35	515.830	0.000	515.830	515.830	1.694	0.176	1.536	2.128
40	571.006	0.000	571.006	571.006	2.454	0.227	2.045	2.931
45	439.870	0.000	439.870	439.870	2.689	0.117	2.503	2.897
50	365.750	0.000	365.750	365.750	3.548	0.237	3.257	3.917
55	1012.729	2.203	1010.930	1015.427	5.958	0.510	5.112	6.717
60	563.660	0.061	563.640	563.844	6.576	0.433	5.633	7.023
65	493.081	0.021	493.074	493.145	7.743	0.682	6.825	9.458
70	441.383	0.000	441.383	441.383	9.732	0.795	8.628	11.389

Table 18. Values of the objective function for the Genetic Algorithm on increasing size of random polygons.

Random distribution

In order to analyze the trend of the solutions and the times of execution, random instances of the “random distribution” problem have been generated with sizes from 5 to 70 nodes with step 5. For each size, 10 different instances of the problem have been solved. Then the mean, standard deviation, maximum and minimum of the series of the recorded values of

the objective function and times of execution have been considered. These results are shown in *Table 19-20-21*.

From *Fig 12* it is clear that both the heuristic methods have proven to be able to find solutions that are reasonably close to the global optimum, with the Genetic Algorithm being always more accurate than Tabu Search. While the Tabu Search is very fast whatever the size, *Fig 13* shows how the Genetic Algorithm starts to be faster than the exact method only after a considerably high size of the instance.

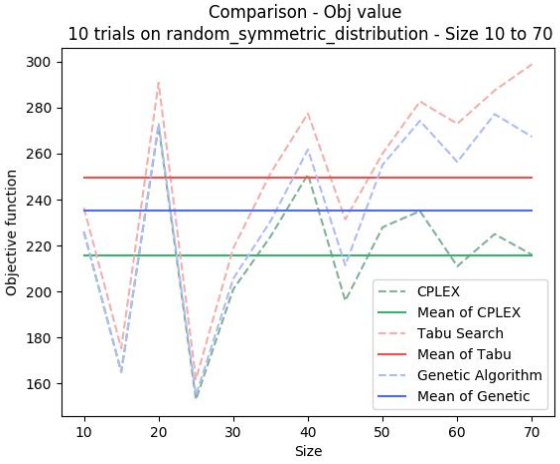

Fig 12. Values of the objective function for 10 trials against increasing size of random distributions, compared to the exact method and the Tabu Search heuristic.

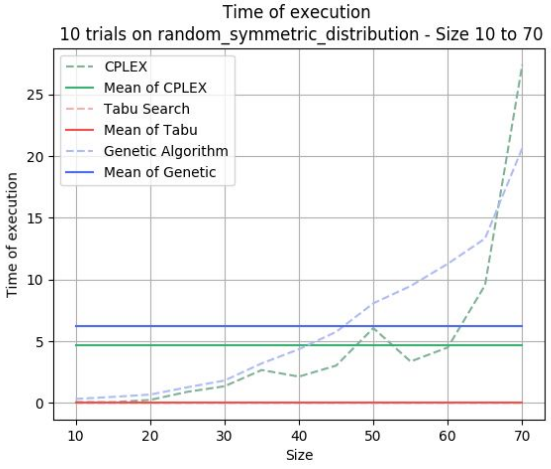

Fig 13. Times of execution for 10 trials against increasing size of random distributions, compared to the exact method and the Tabu Search heuristic.

Exact method via CPLEX

Size	Objective function	Times of execution			
		Mean	Std. Dev.	Min	Max
10	226.000	0.077	0.024	0.028	0.105
15	165.000	0.065	0.006	0.059	0.081
20	272.000	0.246	0.017	0.234	0.283
25	153.000	0.900	0.047	0.865	1.038
30	201.000	1.353	0.434	0.747	2.334
35	224.000	2.670	0.256	2.420	3.123
40	251.000	2.136	0.079	2.006	2.312
45	196.000	3.028	0.497	2.486	4.057
50	228.000	6.069	0.325	5.598	6.503
55	235.000	3.361	0.423	2.759	3.991
60	211.000	4.501	0.176	4.311	4.864
65	225.000	9.542	0.605	9.046	11.040
70	216.000	27.402	2.900	24.198	32.402

Table 19. Values of the objective function and time of execution for CPLEX on increasing size of random distributions.

Tabu search

Size	Values of the objective function				Times of execution			
	Mean	Std. Dev.	Min	Max	Mean	Std. Dev.	Min	Max
10	236.400	12.737	226.000	252.000	1.10e-04	1.79e-05	8.60e-05	1.40e-04
15	175.300	5.950	165.000	185.000	1.72e-04	7.25e-05	9.30e-05	3.32e-04
20	290.900	20.265	272.000	336.000	5.45e-04	3.83e-04	1.18e-04	9.43e-04
25	161.800	10.342	153.000	191.000	8.85e-04	2.85e-04	1.24e-04	0.001
30	218.800	7.467	204.000	232.000	7.26e-04	2.66e-05	6.95e-04	7.73e-04
35	251.500	10.365	234.000	271.000	0.001	2.32e-04	8.49e-04	0.001
40	277.500	7.172	269.000	289.000	0.001	1.43e-04	9.31e-04	0.001
45	231.400	7.748	214.000	243.000	0.001	2.04e-04	0.001	0.001
50	260.100	12.533	243.000	284.000	0.001	4.03e-04	0.001	0.002
55	282.700	9.370	271.000	306.000	0.001	3.25e-04	0.001	0.002

60	273.000	16.558	252.000	311.000	0.001	2.52e-04	0.001	0.002
65	287.500	7.710	277.000	299.000	0.002	2.43e-04	0.001	0.002
70	298.900	21.992	272.000	349.000	0.002	2.47e-04	0.001	0.002

Table 20. Values of the objective function for the Tabu Search on increasing size of random distributions.

Genetic algorithm

Size	Values of the objective function				Times of execution			
	Mean	Std. Dev.	Min	Max	Mean	Std. Dev.	Min	Max
10	226.000	0.000	226.000	226.000	0.327	0.010	0.313	0.345
15	165.000	0.000	165.000	165.000	0.489	0.069	0.408	0.598
20	273.200	1.469	272.000	275.000	0.678	0.106	0.550	0.897
25	154.900	2.844	153.000	161.000	1.269	0.339	0.922	2.014
30	205.600	4.476	201.000	215.000	1.816	0.218	1.400	2.190
35	230.700	3.874	226.000	238.000	3.207	0.519	2.382	4.375
40	261.800	6.584	253.000	270.000	4.356	0.681	3.481	5.646
45	211.400	8.991	200.000	228.000	5.763	1.229	3.444	7.486
50	255.200	11.779	241.000	277.000	8.080	2.301	5.622	12.506
55	274.300	9.829	259.000	294.000	9.479	2.672	5.309	14.063
60	256.400	8.392	245.000	268.000	11.284	2.249	8.295	16.795
65	277.200	12.253	257.000	302.000	13.326	3.051	9.655	19.274
70	267.300	16.180	242.000	293.000	20.657	5.988	13.539	36.577

Table 21. Values of the objective function for the Genetic Algorithm on increasing size of random polygons.

Conclusions

Final results and observations

Analyzing the results of the tests, it is clear how the exact method outperforms the Genetic Algorithm for problem instance of small sizes. However, when the size of the problem becomes bigger than 70 nodes ca. the Genetic Algorithm can find a reasonable approximation of the exact solution faster than CPLEX. Of course, the speed of the Genetic Algorithm highly depends on the value that has been set for the tolerance parameter. The tolerance that was used in the reported tests is 10, which may be lowered losing a bit of accuracy but gaining considerably better times of execution. Analogous reasoning could be made for other parameters such as the size of the population or the mutation percentage.

With regard to the problem under consideration, if the nodes to be considered are at most 20, the obvious choice is that of solving it through the exact method given its greater effectiveness and efficiency. If the number of holes to be drilled is higher, the variability of the configuration of the boards must be taken into consideration. In fact, if each machine was entrusted with the production of a single type of board then it would be better to spend more time to calculate the optimal solution once per machine. Instead, if the boards had different hole configurations to be processed, it would be probably preferable to use a heuristic method, especially if it is necessary for the machine to solve the problem in real-time. In addition, the size of the board and the speed of movement among the nodes must be considered. In fact, the more expensive the manufacturing of the board, the more convenient it is to get a good solution even if this means spending a considerable amount of time. On the other hand, given small boards and fast machines it would be better to focus on saving processing time for the best route, favouring the Genetic Algorithm. Of course, the usage of the heuristic method would also be an obliged choice in case of a limited amount of time or resources to develop an exact algorithm, or if only estimates of the problem parameters were available.

Possible improvements

In conclusion, the Genetic Algorithm seems to achieve good performances in terms of accuracy, while lacking in speed if compared to other heuristics (i.e. Tabu Search in this case). This may be caused by multiple reasons concerning the implementation, which represent the main points where there is room for improvement. First of all, some phases of

the algorithm such as the update of the fitness values, the education and the K-tournament selection could be heavily parallelized. The phase of recombination, in particular, could be refined. As of now, the ordered crossover features a hardcoded $k = 2$ for the k-cut-point. The value k could be calibrated to ensure better recombination. For what concern the selection instead, one could experiment with a linear ranking in place of the K-tournament. This would give the advantage of eliminating one of the parameters of the algorithm. Furthermore, one could implement scheduling of the parameters, meaning that some of the parameters of the algorithm vary at run time based on the current metrics of reference (e.g. the best fitness value in the population or the number of iterations). One aspect of the algorithm that has not been parameterized (but that probably should have) is the percentage of the population that is educated. Reducing the number of educated individuals may cause a sensible reduction of the time of execution, while, on the other hand, lowering the accuracy as well. A possible approach could be that of educating only the elite, fine-tuning the elite size on this basis. Another strategy may consist of actuating an early stopping of the education of an individual as soon as the first improving neighbour has been found. Of course, the k-Opt procedure (now hardcoded with $k = 2$) could be parametrized as well and strengthened with a tabu list. Last but not least, further analysis of the way the initial solution is chosen may have a notable impact on the performances. Choosing a random initial solution avoids biased trends in the values of the objective function, but starting from a fast heuristic solution (e.g. from a greedy solution) should assure a faster convergence.

For what concern the tests that have been conducted, testing on bigger instances of the problem (i.e. with more than 70 nodes) would have produced a deeper analysis of the performances. In particular, it may be interesting to test the here discussed methods on big instances (i.e. hundreds of nodes) offered by public datasets.