Methods and Models for Combinatorial Optimization

Università di Padova

Mirko Bez

February 10, 2017

Contents

Abstract The purpose of this report is to summarize the result obtained during the development of the project for the course "Metodi e Modelli per l'ottimizzazione combinatoria". Given a particular instance of a TSP problem two solutions are provided: an exact one, involving the CPLEX API and a population-based one.

1 Introduction

The aim of this project was to familiarize with different approaches for the solution of a well-known combinatorial problem, especially:

1. An exact solution using the CPLEX API, henceforward referred as problem I, first assignment or exact method (See chapter 3).
2. A metaheuristic approach: a genetic algorithm, from now on referred as problem II, second assignment or genetic algorithm (See chapter 4).

1.1 Problem description

The combinatorial problem to solve was the following:

> A company produces boards with holes used to build electric frames. Boards are positioned over a machines and a drill moves over the board, stops at the desired positions and makes the holes. Once a board is drilled, a new board is positioned and the process is iterated many times. Given the position of the holes on the board, the company asks us to determine the hole sequence that minimizes the total drilling time, taking into account that the time needed for making an hole is the same and constant for all the holes.

This problem can be viewed as a particular instance of the TSP problem: the salesman is represented by the drill and the cities are represented by the holes. Since the problem is NP-hard, the exact method does not halt in a reasonable amount of time when N increases.

Notation In the next sections the formalization of the assignment will be used, in particular N refers to the number of the holes and y_{ij} takes the value 1 if the arc from i to j is used otherwise 0. During the rest of the report the expression *the problem* will be used in order to refer to the given assignment.

Structure The primary focus of the first part of the report concerns with obtaining of instances, in particular with the automatic instance generation. Then follow two chapters about the design choices of the first and second assignment, especially about particular data-structures used. Afterward the attention is put on:

- the performance of the two algorithms,
- the stability of the genetic algorithm

- the quality of the solution provided by this algorithm.

Finally follow a note about the programs usage and the conclusion.

2 Instance Creation

In order to test and compare the performance and the quality of the two algorithms, one fundamental step was to provide some problem instances. In this section I will describe how I obtained the instances used as benchmark.

2.1 Provided Instances

During the class, the professor provides us two instances, with respectively 12 and 60 points.

2.2 Gerber File

In order to represent PBCs there is a standard format, that is called *Gerber*, which extension is `.gbr`. The specification of this file format can be found at: https://www.ucamco.com/files/downloads/file/81/the_gerber_file_format_specification.pdf. This file format contains - among a lot of information (such as lines, drill size, interpolation mode, board size, etc.) - the positions of the drill points.

In order to parse the data from a file in this format, I wrote with my colleague Sebastiano Valle, a parser. It fetches only the points' position. It can be found in the folder `/GerberParser` and can be opened in each browser, because it is written in javascript and is wrapped into an HTML file, named `index.html`. You can upload a file using the button "browse" and it returns a `.dat` file containing the euclidean distance between the holes. The parser tries also to draw the points on the screen, but it works well only with certain instances, because the scale[1] and the center are hard-coded. Therefore in order to view it graphically you can use this online tool http://www.gerber-viewer.com/.

One difficulty with this approach was to find real instances for free on the web. I did not succeed, but my brother provided me a real world example file, that I named `RealWorldExample.gbr`. This instance consists of 53 points. It is used as *real-world* example in the chapter about the result 5 and can be viewed in figure 1.

2.3 Instance Generator

In order to provide some particular instances, I wrote a program that can be found in the folder `/InstanceGenerator`. The usage is straightforward: you have to provide N, the number of points and one letter out of `'r','c','q'` which stands respectively for random, circle, square[2]. The board is square-shaped with a fixed side of $10 \times N$.

- If the option random is provided, it creates randomly N points.
- If the option circle is provided, it creates 4 circles of $\lfloor N/4 \rfloor$ points in the top-left, top-right, bottom-left, bottom-right corner of the drill board. If $N \bmod 4 = a$ with $a \neq 0$ then the circles $0 \ldots a$ will have $\lfloor N/4 \rfloor + 1$ points.

[1]You can zoom in and zoom out.
[2]If no option is provided the random option will be considered.

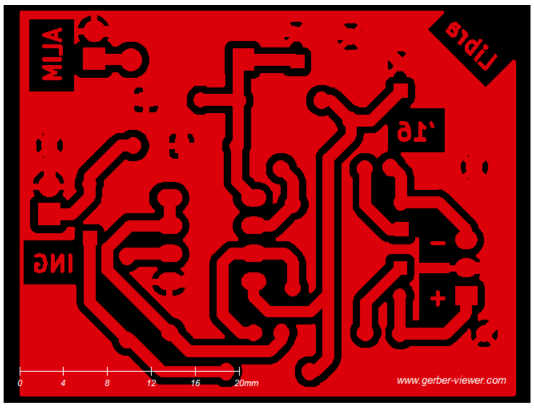


Figure 1: The real world example used.

- If the option square is provided, it creates 4 square of $\lfloor N/4 \rfloor$ points in the top-left, top-right, bottom-left, bottom-right corner of the drill board. If N is not a multiple of 4, it tries to put the extra points somewhere inside the square.

In order to visualize the generated instances, the files `/tmp/tsp_instance_<N>.gbr` and `/tmp/tsp_instance_<N>.pbm`[3] are generated. The latter file can be opened by each image viewer. The instance generator program computes a `.dat` file, that contains the costs from each drill point i to each drill point j. This file can be used as input for the algorithms.

An example of a square and one circle instance can be found in figure 2. These kinds of instances are very similar because the points are clustered on the corners. But the circle instances are more flexible because they work well even with N that are not multiple of 4. Therefore they are preferred over the square ones in the benchmark. The advantage of the circle instances compared to random generated ones is that the first are reproducible and the second are not.

[3]This file format is a good choice because it is portable and easy to encode: 1 bit corresponds to 1 pixel, with 0 = white and 1 = black.

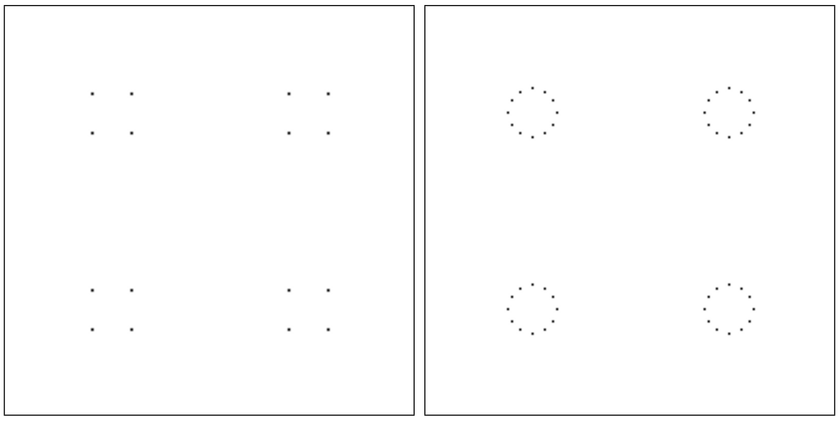


Figure 2: A sample square instance with 16 holes circle instance with 48 holes.

3 First Assignment

The purpose of this assignment was to implement an exact algorithm to solve the problem using the CPLEX API. Before calling the `solve` function it is necessary to set up the CPLEX environment and problem, i.e. to deliver the variables and constraints to CPLEX. This is done by means of the `setupLP` function. Since we are interested (only) in the objective function and the y_{ij} variables, i.e. if the arc from i to j is used, the function takes an additional[4] parameter, `mapY`. This is used to retrieve from the CPLEX API the values of those variables, if the option `--print` is enabled. The set up phase consists in generating the variables and the constraints and passes them to CPLEX.

Variables generation The variables of the problem are x_{ij} and y_{ij} $\forall i, j \in N$. In order to try to render the solution more efficient the program does not pass to CPLEX the variables with index (i, i). Indeed these arcs can be ignored, because they will never be used.

Note about particular Data-Structures In order to render the reference to the generated variables easier during the constraints generation phase, I used two bi-dimensional vectors `mapX` and `mapY`, that associate to the index (i, j) the corresponding CPLEX variables' index. These data-structures are filled during the variables generation phase.

The maps are perfect squares $(N \times N)$, but the entries in position (i, i) are not delivered to CPLEX. The advantage of using this approach is that $2 \times N$ variables are spared (N for x and N for y) and the constraints contain less variables. The drawback is evident, when one write a function that reads the variables back from CPLEX with this configuration. He easily realizes that it is challenging and hard to read. You can check it out in the `fetch_and_print_y_variables` function. Since this functionality was not requested explicitly by the assignment, this drawback can be ignored. Thus this approach is worth the effort.

[4]with respect to the given skeleton.

4 Second Assignment

The second assignment consisted in the implementation of a solution for the problem using a metaheuristic approach. I chose to use a population based solution, because I think that I can reuse this algorithm in the future, because of its great flexibility.

Encoding of the individuals. In order to encode the individuals I used the suggested encoding, i.e. an array of dimension $N + 1$ containing a sequence of holes index, e.g. $< 0\ 1\ \dots\ N\ 0 >$.

Initial Population. The initial population is formed by S individuals. In order to diversify the population, fast all the individuals are chosen randomly, but the increasing sequence $< 0\ 1\ \dots\ N\ 0 >$ and the decreasing sequence $< 0\ N\ \dots\ 1\ 0 >$ are always granted, in order to have a couple of solutions with big Hamming distance.

Fitness Function. For simplicity the fitness function is the inverse of the objective value.

Selection. Although I had implemented the three methods showed in class, I used the *Montecarlo* approach for the selection during the increasing phase, because this method leads to a very fast convergence; instead in the diversification phase the *N-Tournament* (with size R) approach was preferred , because it mitigates the effects of the chosen population substitution strategy, that keeps only the best elements.

Combination. For the combination I used the ordering crossover with two randomly chosen cutting points by adopting the `std::map<int,int>` data-structure, defined in the C++ header `<map>`. This data-structure has two desirable features:

- It is very efficient because it is implemented as a RB-Tree[5]
- More important, it is sorted by key, i.e. the first element of the pair.

The latter characteristic permits an easy implementation of the ordering crossover.

The efficiency of the Combination cannot be neglected, because this operator is used R times in each iteration and in the worst case the difference between the two randomly chosen cutting point can be very close to N. The usage inside the algorithm of this data-structure can be viewed in the file `TSPSolverGA.h`.

Intensification and Diversification The algorithm starts with an intensification phase. After 500 non improving iterations, a diversification phase follows during which

- The mutation is performed with a greater probability;
- The selection is done according to the n-tournament strategy;

[5] http://en.cppreference.com/w/cpp/container/map

- The values are substituted randomly but the algorithm keeps always the best two elements of the population.

Population Substitution. During each iteration R new individuals are created, and attached to the end of the population. Therefore the cardinality of the new population is $S + R$. Afterward the population is sorted in descending order according to the fitness value.

Since R is smaller than S and the population at the start of the iteration is sorted, *insertion sort* is used. This works very efficiently (linearly) with inputs that are almost sorted. Instead the standard sort algorithm is (usually) *quicksort*, which performs extremely poor (in quadratic time) with almost sorted vectors.

After the sort operation only the S best elements are kept, discarding the last R elements. Instead during the diversification phase, this strategy is substituted by a random choice of elements to keep, but it is always granted that the best two elements are kept in the population.

Stopping Criterion. The algorithm terminates when it reaches 500 non improving iterations after the alternation of two intensification and two diversification phases. The drawback of this approach compared to a time based stop is that it is not predictable when the algorithm will finish. In addition sometimes the improvements are so low that it is not worth to repeat other 500 iterations.

Finalization At the end of the algorithm the best element[6] of the population is returned. But first it is trained with a random greedy local search, that render it (possibly) even better.

4.1 Parameter calibration

Since the factors on which the solution depends are many - among others the probability of the mutation, selection method, number of non improving iterations, size of population, how the initial solution are created etc - I fixed the majority of them after a number of tries. In the section 7 the impact of the size of the population on the quality and performance of the algorithm will be discussed.

[6]It corresponds to the first element of the population, because it is sorted.

5 Result

Inside this section the principle results of my work are illustrated. Note that the absolute performance values depends strongly on the hardware of the machine used. All the tests were run on a Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz.

In order to show the performance and the differences between the two implementations I selected some files that can be found in the `/Sample` directory.

Each instance was run ten times and the average user time was computed. This measure was preferred over the cpu time, for two reasons:

- because the CPLEX solution is multi-threaded it requires circa 6-7 times more cpu time than the one-thread genetic algorithm.
- a company that produces drill boards is more interested in the wall clock time than the cpu one.

5.1 Benchmark description

The instances used during the test are the following:

- TSP12, the tsp instance with 12 elements given during the class;
- TSP60, the tsp instance with 60 elements given during the class;
- REALWORLD, the real-world tsp instance, with data extrapolated from a real-world `.gbr` file mentioned above in section 2.2.
- CIRCLE3-50 Circle instances 3 to 50 (See chapter 2.3)
- RANDOM3-50 50 random but fixed instances, used in order to show the general-purpose efficiency of the genetic algorithms and the quality of the provided solutions;
- RANDOM100a and RANDOM100b, two random but fixed instances in order to show the general-purpose efficiency of the genetic algorithm and observe that the exact solution is well-suited for (even reasonably big) random instances.

During the rest of the paper the names of the tests will be fixed.

6 Result - First Assignment

In this part of the report the focus is mainly on the performance of the exact solution. In general the variance between the different exact solution runs is negligible and therefore these data are not reported.

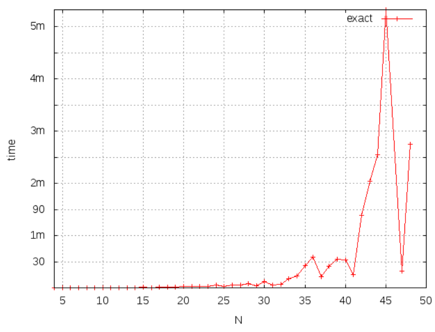
Now follow the considerations about the single benchmarks:

CIRCLE3-50 The results provided by this benchmark can be seen in figure 3a. They are quite surprising because the time does not grow (always) proportionally with the increase of N. Note especially the entries 37,41,47 require a very small amount of time. Instead there are also inputs with which CPLEX cannot deal easily, such as 46, that was not reported in the graph because it required 23 minutes, 49 that is not reported because after circa 1 hour it did not return back a result.

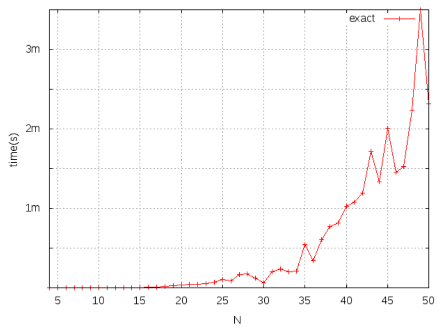
RANDOM3-50 The result of this benchmark (figure 3b) shows that, when N increases, the required time grows accordingly, especially from 36 on. Even though there are some exceptions, because of the lack of a strong relationship between the random instances, the trend is evident.

TSP12, TSP60, RealWorld The TSP60 and the RealWorld examples consist of circa the same number of points and therefore the required time lies in the same order of magnitude as shown in figure 4b.

Random100a, Random100b The results can be viewed on figure 4a. They show that the exact algorithm works well particularly with random instances.



(a)

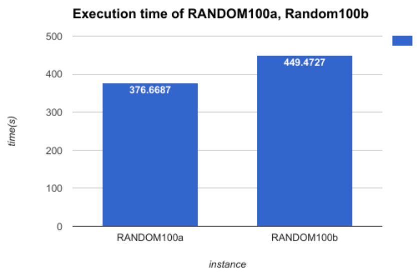


(b)

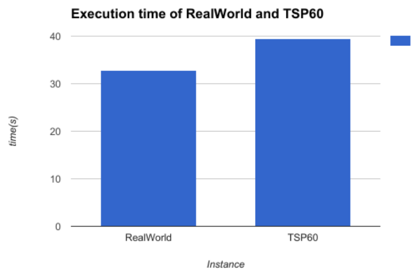
Figure 3: The time result of the benchmark CIRCLE3-50, RANDOM3-50

Considerations. In general from the above graphs I concluded:

- That with a small N (from 3 to 30) the program performs blazingly fast.



(a)



(b)

Figure 4: The time result of the benchmark Random100a and Random100b

- The program works better with solutions that are generated randomly, i.e. with points distributed across the whole board. Instead it requires (generally) more time when the points are grouped in the corners of the board. In this case we can observe that the performance of the exact algorithm becomes very unstable, you can consider $N = 46$ and $N = 47$ as an example (See figure 3a).

Instance	Time CV - 10	AVG - 10	Time CV - 100	AVG - 100	ratio
TSP12	31.78%	0.04	24.62%	0.40	10
TSP60	13.77%	0.52	28.77%	4.55	8.75
REALWORLD	9.50%	0.30	28.25%	1.83	6.1
TSPR100a	20.70%	1.25	24.72%	10.64	8.51
TSPR100b	20.33%	1.23	18.84%	11.56	9.39
Average	19.2%		25.04%		8.55

Table 1: Comparison about the time performance, when using 10 or 100 as population size.

Instance	Value CV - 10	AVG - 10	Value CV - 100	AVG - 100	ratio
TSP12	0.0%	66.4	0.0%	66.4	1.0
TSP60	6.64%	836.54	5.04%	832.99	0.995
REALWORLD	4.34%	2.94E+08	4.93%	2.83E+08	0.963
TSPR100a	6.37%	16,726.31	7.39%	16,872.59	1.008
TSPR100b	8.75%	16,320.84	9.67%	16,728.37	1.024
Average	5.22%		5.40%		0.99

Table 2: Comparison about the stability of the algorithm, i.d. in the variance of the values and average values.

7 Result-Second Assignment

This section is divided into two parts: the first analyzes the time required by the algorithm and the stability of the results provided with different configurations for S, the population size.

This algorithm relies on the random choices made during the execution and on other factors such as the population size.

Note about the tuning of the algorithm Note that R, the number of new individuals generated in each iteration, is dependent on S. It is obtained as the ceiling of the square root of S[7]. This implies that $R << S$ and therefore the selection sort works excellently.

In order to choose the population size I tested TSPRANDOM100a-b, TSP12, REALWORLD and TSP60 10 times with population 10 and population 100. The main results are summarized in the tables 2 and 1. As you can see in the last line of table 2 the standard deviation of the values, is very low. Since the calculated average value are substantially equivalent the population size 10 was preferred for its efficiency, that is 7-8 times faster than the other configuration.

[7]If this number is odd it is incremented.

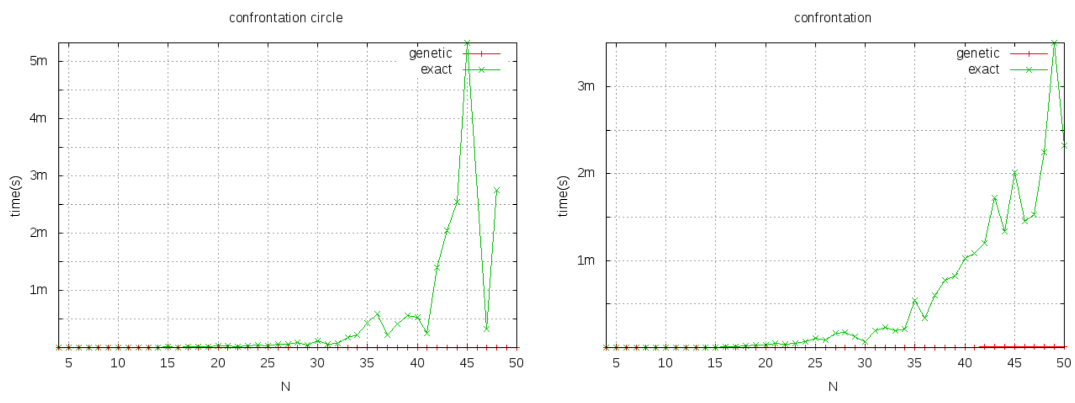


Figure 5: Comparison between exact and genetic algorithm on a time basis.

Instance	Exact Solution	AVG Error	Worst Case	Best Case
TSP12	66.4	0%	0%	0%
TSP60	629.8	32.83%	42.60%	17.99%
REALWORLD	222811276	31.89%	37.65%	24.32%
TSPRANDOM100a	9778.91	71.04%	87.75%	51.17 %
TSPRANDOM100b	9802.71	66.50%	83.28%	42.50%

Table 3: In this table the result of the comparison are formalized. The coefficient of variation is used to show the deviation between the 10 computed result.

8 Comparison between Solution I and Solution II

It does not make too much sense to compare the metaheuristic and the exact approach on a time based, because the genetic algorithm is tremendously faster as described in figure 5.

8.1 Quality of the result

Inside this section the result provided by the genetic algorithm are compared with those generated by the exact solution. In the table 3 the results given by the genetic algorithm are compared with the optimal solutions. As seen in the previous section the algorithm is quite stable and with small instances it can provide very good solutions compared to the optimal solutions, as summarized in the table 3 with an error of circa 20 %. it can find good values compared to the optimal solutions, as summarized in the table 3. But with the increasing of N the precision becomes worse and worse.

9 Usage/Practical Notes

In this section the usage of the programs are briefly described. See section 2.2 for the generation of instances from files in gerber format. Details about the usage of the instance generator can be found in section 2.3.

First Assignment In order to compile the first assignment a `Makefile` is provided. After the compilation the file `main` is created. It needs an instance in `.dat` file format that can be found in folder `/Sample`, in which the first line contains N. The remaining N rows contain N entries in which the value (i, j) represents the distance from drill point i to drill point j. The program can be run with the following command:

```
./main <tsp_instance.dat> [OPT]
```

In order to view the available options you can use the `--help` option. The only option that is worth it to mention here is the `--benchmark` one. This causes the suppression of useless messages and formats the output as well.

Second Assignment The solution for the second assignment can be found inside the `/ExamExercise2/Genetic` folder. A `Makefile` is provided. The program can be run with following command:

```
./main <tsp_instance.dat> [Population Size]
```

Optionally one can specify the population size as a third argument.

Program Documentation In the program folders you can also find a file named `Doxyfile` that can be used to generate the documentation of the programs using the tool `Doxygen`.

10 Conclusion

Inside this report it was shown that CPLEX can work very efficiently with small instances and random instances in general. During the measures it turns out evident the CPLEX perfoms in general worse and in an unpredictable way, when the holes are cluestered in the corners of the board.

The tuning and analysis of the efficiency and stability of the population based solution was extremely simplified by fixing the majority of the parameters. The genetic algorithm is very fast and provides a solution in brief time even if the input is big, especially when the population size is set to 10. With the increasing of the problem size the precision of the genetic algorithm becomes worse and worse.

In general I can state that with a little problem size it does not make sense to write and try to optimize a genetic algorithm.

I assume that a company that produces PBCs will replicate the given board millions of time. Considering that, even if it will take a lot of time to the producer to get the exact solution the first time, at the end they will be much faster. The initial investment will pay off rapidly.

In general this kind of companies do not produce boards with several hundreds of holes, but more likely with less than 100[8]. The user time reported refers to a five years old laptop, but a company can surely rely on a better hardware.

[8]At least in the real world example.