Methods and Models for Combinatorial Optimization

Università di Padova

Mirko Bez

February 9, 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 **??**
2. A metaheuristic: a genetic algorithm and simulated annealing were implemented. **??**

1.1 Problem description

The assignment of the problem 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, i.e. with inputs bigger than 60.

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 reference. to the assignment.

Structure In the first part of the report the focus is put on instance creation, design choices of the first and second argument, particularly about the data-structures used, in the second part the attention is put on the performance and precision confrontation.

2 Instance Creation

In order to test and compare the performance of the algorithms, one fundamental step was to provide some instances. In this section I will describe how I obtained th instances:

2.1 Provided

During the class, the professor provides us two instances.

2.2 Gerber File

In order to represent PBC 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. The parser - written with the collaboration of my colleague Sebastiano Valle- parses only the points' position. It can be found in the folder `GerberParser`. It can be opened in each browser, because it is written in javascript wrapped in an HTML file, index.html. You can upload a file using the button "browse" and it returns a `.dat` file containing the euclidean distance between the holes, it tries also to draw the points, but it works well only with certain instances, because the size of the screen 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 example file, that I named `RealWorldExample.gbr`. This instance consists of 53 points it is used as *real-world* example in the test 5 and can be viewed in figure 2.2.

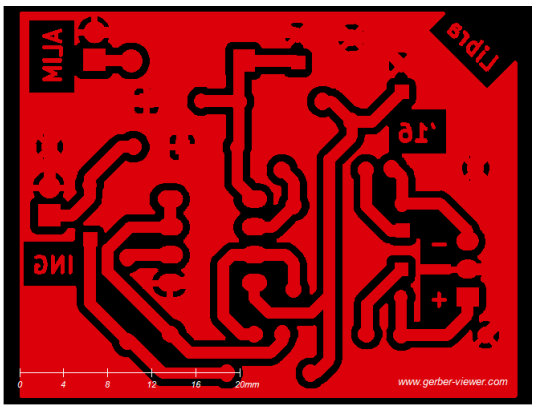


Figure 1: The real world example used.

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[1]. The board is square-shaped with a fixed size of $10 * N$.

- 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 $0..a$ will have $\lfloor N/4 \rfloor + 1$ points.
- 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`[2] are generated. The latter file can be opened by each image viewer. The instance generator program computes also a `.dat` file containing the costs from drill point i to drill point j.

The square and circle instances are very similar because they create instances in which 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 used in the benchmark. The advantage of the circle instances with respect to random generation is that the are reproducible the second not.

[1] If no option is provided the random option will be considered.

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

3 First Assignment

The purpose of this assignment was to implement an exact algorithm to solve the problem using the CPLEX API. In order to call the solve function of the CPLEX library, previously I have to set up the environment and the problem, i.e. pass the data to CPLEX. This is done by means of the `setupLP` function. Since we are interested (only) in the objective function and y_{ij} variables, i.e. if the arc from i to j is used, the function takes an additional[3] parameter, `mapY`. It 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 gives them to CPLEX.

Variables and constraints 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 Data-Structure In order to render the reference to generated variables easier during the constraint generation phase, I used two bi-dimensional vector `mapX`, `mapY` that contains in position ij the corresponding CPLEX-variable index. They are filled during the variable generation phase.

The maps are perfect squares ($N \times N$), but the entries in position i, i are not passed to CPLEX. The advantage of using this approach is that $2 * N$ 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.

[3] w.r.t. to the given skeleton

4 Second Assignment

The second assignment consist in the implementation of a solution for the problem using a metaheuristic approach. I chose to use a genetic algorithm, 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. $< 0\ 1\ \ldots\ N\ 0 >$

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 $< 0\ 1\ \ldots\ N\ 0 >$ and the decreasing sequence $< 0\ N\ \ldots\ N\ 0 >$ are always granted, in order to have solutions with big distance. In order to start with better solution some of them are trained with a random local search.

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

Selection. Although I implemented the three methods showed in class, I used *monte-carlo* in the increasing phase, because this method leads to a fast convergence; instead in the diversification phase I used the linear-ranking approach, because it mitigates the effect of the population substitution strategy, that saves only the best elements.

Combination. For the combination I used the ordering crossover with two randomly chosen cutting points. To perform this kind of crossover I adopted the `std::map<int,int>` data-structure, defined in the header `<map>`. This data-structure has very good performance[4], because it is based on RB-Tree and most importantly it is sorted by key, i.e. the first element of the pair. The efficiency of the Combination cannot be neglected, because this operator is used R times in each iteration and in the worst case the size of the cut point can be very close to N. The usage inside the algorithm can be viewed in the file `TSPSolverGA.h`.

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

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

After the sort operation only the N best elements are kept, discarding the last R elements.

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

Stopping Criterion. The algorithm terminates when it reaches 500 non improving iterations. The drawback of this approach 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, but otherwise it could happen that a few more iteration would have produce better result. The advantage w.r.t. the stop after a while is that it is not arbitrary.

Finalization At the end of the algorithm the first element of the population is returned. But first it is done a greedy local search on the best value in order to render it even better. Because he has already good characteristics, the local minimum could correspond with the optimal one.

4.1 Parameter calibration

5 Result

The following results are depending also on the hardware of the machine used. The following tests were run on a Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz. In order to show the differences between the two implementations I selected some files that you can find in the `Sample` directory. I wrote a script `benchmark.sh` that runs each instance of the problem ten times and then it computes the average user time[5]. In order to test this problem, I used the following instances

- 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 `.gbr` file mentioned above;
- CIRCLE3-50 Circle instances 3 to 50
- RANDOM3-50 50 random but fixed instances, used in order to show the general-purpose efficiency of the genetic algorithms;

During the rest of the paper this names will be used.

5.1 Result - First Assignment

In this part of the port the focus is put on the performance of the exact solution. The result of the exact solution can be found in the corresponding images

CIRCLE3-50 The result of this benchmark are quite surprisingly because the time does not grow proportionally with the increase of N. Note especially the entries 37,41,47 requires a very small amount of time. Instead there are also entries that do not work good, such as 49 and 64[6].

RANDOM3-50 The result of this benchmark shows that when N increases the time required grows accordingly, but there are some exceptions.

TSP12, TSP60, RealWorld The TSP60 and the RealWorld example consists of circa the same number of points and therefore required a time in the same order of magnitude.

In general from the above graphs I concluded:

- That with a small N (from 3 to 30) the program performs very well

[5] In general the CPLEX API cpu time is much bigger(7 time) than the genetic algorithm time because CPLEX is multithreaded, the implementation of the second algorithm use only one thread

[6] It was excluded because even on the lab computer it required more than 42 minutes (I did not a get a result)

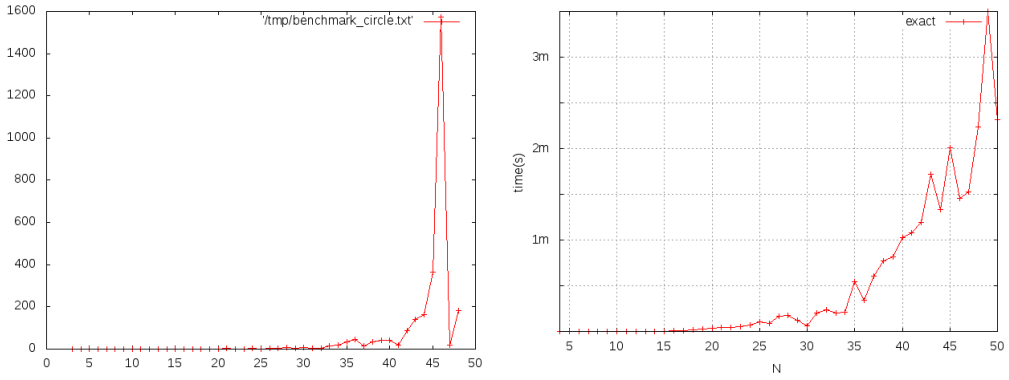
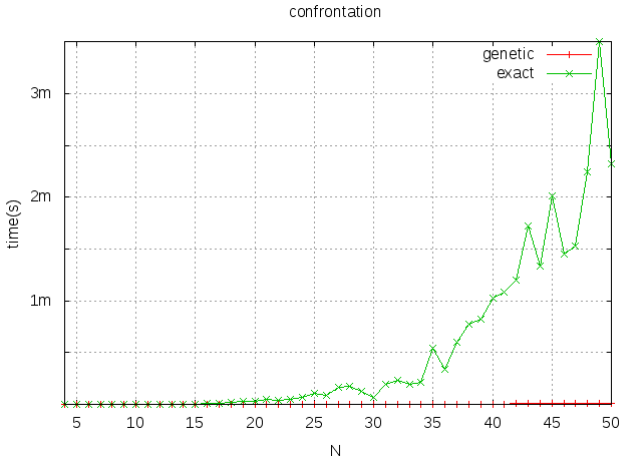


Figure 2: The time result of the benchmark CIRCLE3-50, RANDOM3-50

- 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. More importantly when this is the case the performance of the CPLEX API becomes very unstable (you can consider $N = 46$ and $N = 47$ as an example).



6 Result - Second Assignment

7 Result-Second Assignment

The time required by the second assignment is not so trustworthy as the first one, because it depends strongly about the random generated population. Therefore this section is divided in two parts: the first treats the time required by the algorithm and the second one compare the objective value provided by the exact solution with the one found with the genetic algorithm. During the 10 run I take the best the worst and the average one.

7.1 Performance

As already mentioned this approach relies on the random choices made during the execution but also on the population size, the number of elements generated. In general it is much faster as you can see in the graphs 7.1 and therefore it does not make sense to compare the first and the second algorithm on the basis of efficiency.

7.2 Quality of the result

Inside this section the result provided by the genetic algorithm are compared with those generated by the exact solution. Each graph contains the average result.

8 Comparison between Solution I and Solution II

It does not make to much sense to compare the metaheuristic and the exact approach on a time based, because the genetic algorithm is faster.

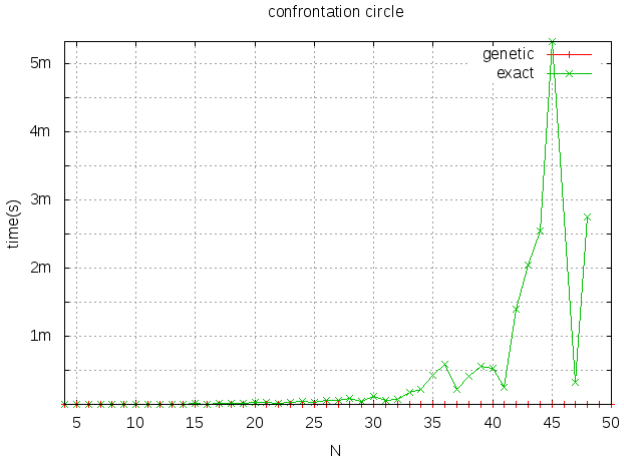


Figure 3: N.B. the input 46 was removed because it required more than 23 minutes.

9 Usage

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, in which the first line contains N, and the remaining N contains N entries in which the entry i, j represent the distance from drill point i to drill point j.

```
./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` option that suppress useless messages and format the output, so that the `benchmark.sh` can write the information in a text file that can be plotted using `gnuplot`.

Second Assignment

10 Conclusion

It turns out that CPLEX with problem size bigger than 45 becomes very slow. For instance with a circle problem of size 64 I stopped the program after 42 minutes. From my point of view one of the most interesting test was the real-world test.