Methods and Models for Combinatorial Optimization

Università di Padova

Mirko Bez

February 9, 2017

Contents

Abstract Report about the project of the class "Metodi e Modelli per l'ottimizzazione combinatoria".

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.

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.

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`. In order to view it "graphically" you can use this online tool http://www.gerber-viewer.com/. This instance consists of 53 points, it is used as *real-world* example in the test**??**.

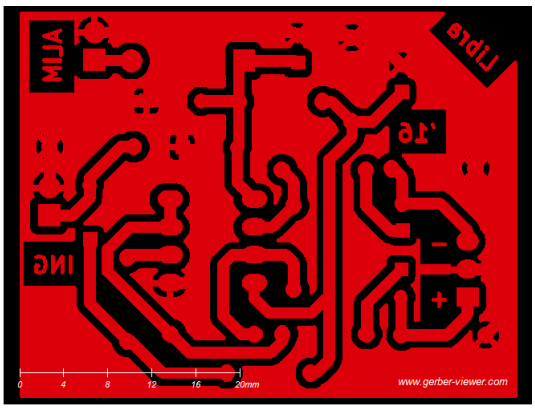

Figure 1: The real world example used.

2.3 Istance Generator

In order to provide some particular instances, I wrote a program that can be found in the folder `InstanceGenerator`. The usage is very simple: you have to provide N, the number of points and one letter out of `r,c,q` which stands respectively for random, circle, square:

- If the option random is provided, it creates randomly N points in a square-shaped board of dimension $2 * N$.
- 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 instances the files `/tmp/tsp_instance_<N>.gbr` and `/tmp/tsp_instance_<N>.pbm` 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.

[1] 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[2] 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 costraints 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 bidimensional 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 by the assignment, this drawback can be ignored. Thus this approach is worth the effort.

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

4 Second Assignment

In this second assignment the purpose was to implement a solution for the problem using a metaheuristic approach. I chose to use a genetic algorithm TODO aggiungi motivazoione

Encoding of the individuals. In order to encode the individuals I used the provided 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 k 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 start with better solution some of them are trained.

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. In order to perform this kind of crossover I use the `std::map<int,int>` data-structure defined in the header `<map>`. This data-structure has very good performance[3] and most importantly it is sorted by key, i.e. the first element of the pair.

Population Substitution. During each iteration R new individuals are created. 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 with already sorted vectors $O(N + R)^2$. Therefore only the first N elements are kept, discarding the last R elements.

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.

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

4.1 Parameter calibration

5 Result

The following results are depending also on the HW 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 user time[4]. 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;
- CIRCLE48, an instance generated with the InstanceGenerator that is formed by four very distant circles composed by 12 elements each;
- CIRCLE3-45 Circle instances 3 to 45;
- RANDOM1 a random instance, in order to show the general-purpose goodness of the algorithm;
- RANDOM2 a random instance, in order to show the general-purpose goodness of the algorithm;

5.1 Result - First Assignment

6 Result - Second Assignment

[4] 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

7 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.