Memoco

February 8, 2017

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 approach for the solution of a well-known combinatorial problem, in particular the assignment required to solve the problem with two different approaches:

1. An exact solution using the CPLEX API https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/??
2. A metaheuristic. ??

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, in which the salesman is represented by the drill and the cities are represented by the holes. Because of the NP-Hardness of this problem it is a good candidate for an implementation of a metaheuristic approach, particularly when the size of the problem, i.e. the number of holes becomes bigger.

1.2 Problem Formalization

- y_{ij} 1 if the edge between hole i and j is used.

2 Instance Creation

To test the performance of the algorithms and compare them one fundamental step was to provide some ”good instances”. In this section I will describe how I obtain the isntances:

2.1 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 format contains - among many (for us not useful) 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 ”upload” 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 provides 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/, it consists of 53 points.

2.2 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 between `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 see if the instances created are well-formed and are exactly what we would expect the program writes in the files `/tmp/tsp_instance_<N>.gbr` and `/tmp/tsp_instance_<N>.pbm`[1] the points in the specified format. The latter file can be opened by each image viewer.

[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

The instance generator program compute also a `.dat` file containing the costs from drill point 1 to drill point 2.

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 (wrt to the given skeleton) parameter, `mapY` that is used to eventually (if the option `--print` is enabled) retrieve from the CPLEX API the values of those variables. The set up phase consists in generating the variables and the constraint to CPLEX.

Variable 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 easier to refer to generated variables, i.e. the x_{ij} and y_{ij} variables, during the constraint generation phase, I used two 2D 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 square ($N \times N$), 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 constraint contains in general less variables. The drawback is evident, when one try to write a function that reads the variables back from CPLEX, in fact it is challenging to write and hard to read as you can see in the function `fetch_and_print_y_variables`. Since this functionality was not requested, the CPLEX computational saving did not bring with it much drawbacks.

4 Second Assignment

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

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 montecarlo in the increasing phase and linear-ranking in the diversification phase, in order to mitigate the effect of the Population Substitution.

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[2] and most importantly it is sorted by key, i.e. the first element. It is a RB-Tree. With

Population Substitution During each iteration R new individuals are created. Therefore the new population is of size $N + R$, but only the N best are mantained the other R are discarded. This is achieved by sorting the population by increasing value (i.e. decreasing fitness).

Stopping Criterion The algorithm terminates when it reaches 500 non improving iterations, i.e. iterations in which there was no improvement.

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