Report

Methods and Models for Combinatorial Optimization

Student: Brugnolaro Filippo

ID: 2087666

University of Padova

A.Y. 2023/2024

Contents

1 Introduction

A company produces boards with holes used to build electric panels. Boards are positioned over a machine 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.

1.1 Mathematical Formulation

This problem can be seen as a *Traveling Salesmen Problem (TSP)*. We can represent it on a weighted complete graph $G = (N, A)$, where N is the set of nodes representing the positions of the holes to be drilled and A is the arcs making the trajectory from one hole to the another. The optimal solution is the Hamiltonian cycle on G which has the minimum weight. The *TSP* can be formulated also as a network flow model on G, hence we use this reduced mathematical formulation of the problem:

Model:

$$\min \sum_{i,j:(i,j)\in A} c_{ij} y_{ij} \tag{1}$$

$$\text{s.t.} \sum_{i:(i,k)\in A} x_{ik} - \sum_{j:(k,j),j\neq 0} x_{kj} = 1 \quad \forall k \in N \setminus \{0\} \tag{2}$$

$$\sum_{j:(i,j)\in A} y_{ij} = 1 \quad \forall i \in N \tag{3}$$

$$\sum_{i:(i,j)\in A} y_{ij} = 1 \quad \forall j \in N \tag{4}$$

$$x_{ij} \leq (|N| - 1) y_{ij} \quad \forall (i,j) \in A, j \neq 0 \tag{5}$$

$$x_{ij} \in \mathbb{R}^+ \quad \forall (i,j) \in A, j \neq 0 \tag{6}$$

$$y_{ij} \in \{0, 1\} \quad \forall (i,j) \in A \tag{7}$$

where sets, parameters and decision variables are defined as:

Sets:

- N = graph nodes, corresponding to the holes positions;
- A = arcs (i, j), $\forall i, j \in N$, serving as the movement from hole i to hole j.

Parameters:

- c_{ij} = time to move from i to j, $\forall (i, j) \in A$;
- 0 = arbitrary starting node, $0 \in N$.

Decision variables:

- x_{ij} = amount of the flow shipped from i to j, $\forall (i, j) \in A$;
- y_{ij} = 1 if arc (i, j) ships some flow, 0 otherwise, $\forall (i, j) \in A$.

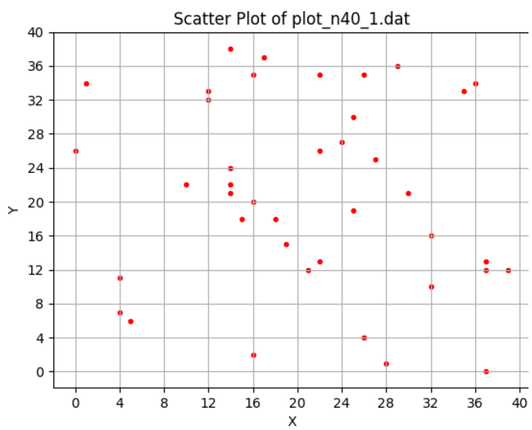
1.2 Input representation

An instance of the problem is represented as a list of points on the Cartesian plane. The first line of an input instance contains an integer n, representing the size of the instance or more precisely the number of points. Then each of the following n lines contains two integers separated by a space, respectively representing the coordinates (x, y).

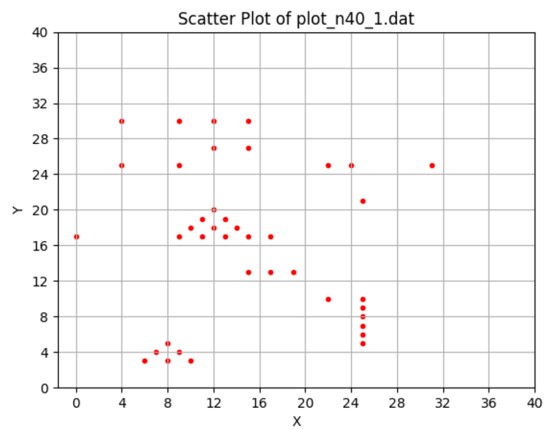
1.3 Instances generator

The class *instancesGenerator* deals with generating an instance of the problem. The generator can implement 2 types of instances, as represented in Fig.1:

- **random:** generation of points in the matrix;
- **shapes:** generation of random shapes in the matrix with a *uniform distribution* of choices between squares, triangles, rectangles and vertical/horizontal lines.



(a) Instance with random points



(b) Instance with shapes

Since duplicates of points were a possibility, a *std::set<std::pair<int, int>>* has been used. Furthermore, it is possible to see from code in Listings 1 and 2 that random_device and mersenne_twister_engine are used to create a generator with non-deterministic seed used as function for both *uniform_int_distribution* instance and *shuffle* method. For more informations read the *<random>* library documentation.

```
std::random_device rd;
std::mt19937_64 generator(rd());
std::uniform_int_distribution<int> choice_distr(0, 4);
int choice = choice_distr(generator);
```

Listing 1: Figure choice generation

```cpp
std::vector<std::vector<double>> allPos;
// ... allPos initialization
std::random_device rd;
std::mt19937_64 generator(rd());
// shuffle all pairs
std::shuffle(allPos.begin(), allPos.end(), generator);
```

Listing 2: Random points generation with all positions' shuffle

1.4 Problem representation

In this analysis, some assumptions have to be taken into consideration:

- let n be the size of the instance, then the n points are included in a nxn matrix;
- the position of an hole i in the motherboard is represented by a pair (x_i, y_i);
- let i, j be 2 points of the instance, the cost of the arc (i, j) is calculated as the *Manhattan distance*, i.e sum of the modulo's differences between the corresponding coordinates, $c_{ij} = |x_i - x_j| + |y_i - y_j|$;
- the solution of the problem is represented as a list of integers, i.e. list of indexes of points, and the first and last element are repeated, i.e. the start and the end of the cycle.

Example

The instance in Fig. 2 corresponds to the following points:

$$(9, 7); (4, 1); (1, 3); (6, 0); (3, 4); (1, 8); (8, 9); (7, 6); (5, 3); (8, 7)$$

and each point is mapped with an index for each $i \in A$, as explained in Section §2.2.

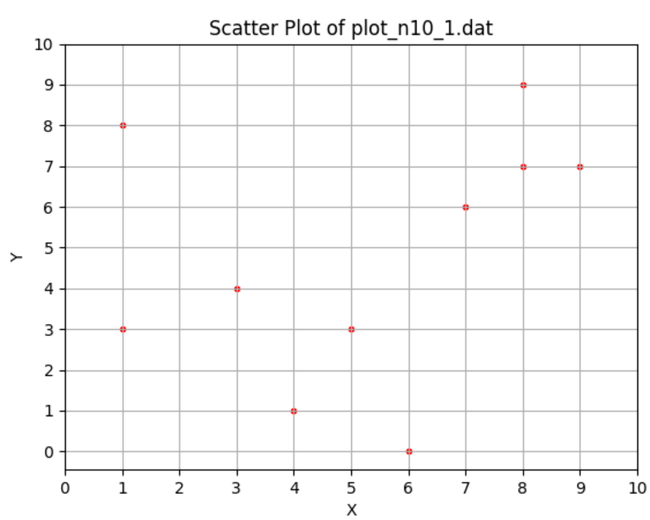


Figure 2: Graphical representation of an instance

The weighted adjacency matrix of the instance can be represented as follows:

x	0	1	2	3	4	5	6	7	8	9
0	0	11	12	10	9	9	3	3	8	1
1	11	0	5	3	4	10	12	8	3	10
2	12	5	0	8	3	5	13	9	4	11
3	10	3	8	0	7	13	11	7	4	9
4	9	4	3	7	0	6	10	6	3	8
5	9	10	5	13	6	0	8	8	9	8
6	3	12	13	11	10	8	0	4	9	2
7	3	8	9	7	6	8	4	0	5	2
8	8	3	4	4	3	9	9	5	0	7
9	1	10	11	9	8	8	2	2	7	0

Table 1: Weighted adjacency matrix of an instance

A solution of the problem could be represented as:

$$[0, 4, 3, 7, 5, 9, 2, 6, 8, 1, 0]$$

2 CPLEX

The model is implemented and solved with the *CPLEX APIs*. The following sections describe some implementation details.

2.1 Rows and columns generation

Rows and columns are created one at a time by the *setupLP()* method in which *CPLEX APIs* are called through the macro *CHECKED_CPLEX_CALL* in order to manage exceptions and errors. Model implementation is straightforward, but there is one constraint that requires some attention and form of reasoning.

$$\textstyle\sum_{i:(i,k)\in A} x_{ik} - \sum_{j:(k,j),j\neq 0} x_{kj} = 1 \qquad\qquad \forall k \in N \setminus \{0\}$$

Since the variables x_{kk} appears on both sums, it is useful to avoid adding its index twice. Since the coefficients of the variable in the different sums are opposite, i.e. 1 and -1 respectively, we can just ignore $x_{kk}, \forall i \in N$. The following code represents the operation:

```
for (int i = 0; i < N; i++) {
    if (i == k) continue;        // exclude x_kk
    idx.push_back(map_x[i][k]);  // corresponds to variable x_ik
    coef.push_back(1.0);
}
```

Listing 3: Avoiding x_{kk}

2.2 Mapping variables

In order to store the indexes of the variables, the *std::vector<int>* data structure is used. Specifically, two vectors are employed, one for the x-values and another for the y-values, and this implies a time complexity and space complexity of $O(n^2)$.
There was also the possibility to use a *std::map*, but complexities of operations were higher than *std::vector*, since *std::map* is implemented as a binary search tree, e.g. access and adding a value in $O(log\ n)$, and this would have implied a time complexity of $O(n^2\ log\ n)$.
It is important to underline that the model generation is not a critical point for the *CPLEX* method's results, since the difference between $O(n^2)$ and $O(n^2\ log\ n)$ is not relevant.

3 Tabu Search

The problem is also solved with the *Tabu search* method. The implementation uses the same instances obtained with *CPLEX*, so there is not any generation of new instances but only reading operation from weighted adjacency costs.
The following sections describe some implementation details and design choices, taking inspiration also from the implementation of *TSAC* done during the course.

3.1 Tabu list

One of the most important things of a Tabu search metaheuristic is the tabu list's implementation. The aim is to understand whether the move the algorithm is going to do is too much recent or not. A *std::vector<int>* has been used to store all the n nodes as indexes and the iteration of the last successful move for that node as value. Using this approach, updating the tabu list or checking whether the current move is tabu or not (Listing 4) have both $O(1)$ time complexity for each iteration.

```
bool TSPSolver::isTabu(int nodeFrom, int nodeTo, int iter) {
    return ((iter - tabuList[nodeFrom] <= tabuLength) &&
    (iter - tabuList[nodeTo] <= tabuLength));
}
```

Listing 4: Checking tabu move

3.2 Initial solution

In order to increase the methods' performances, it is necessary to have a good initial solution. In fact, it can largely influence the tabu search results in terms of quality of solution or convergence speed. In particular, two types of initialization have been created:

- **random:** create a random initial solution by taking at each iteration 2 indexes and swapping them;
- **nearest neighbour:** create an initial solution by taking the minimum cost between the current node and the next non-visited node.

The principal aim is to understand how much the two different approaches influence the initial solution and how performances can vary depending on it. Benchmarking results will give a more precise idea and will be shown in Section §4

3.3 Parameters

The parameters that can be calibrated are the following:

- tabu-list length;
- initial solution;
- stopping criteria.

The following stopping criteria has been implemented:

- max iterations;
- max non-increasing iterations;
- time limit.

Their implementation is straightforward and there are not any interesting design choices.

4 Computational results

In this section there are the computational results of both *CPLEX* and tabu search. It is important to underline the fact that a test for $N = 200$ has been done with *CPLEX*, but we obtained very high executions time (4153.88 and 12365.8 seconds). Then also for the tabu search the runs were taking too much time and for that reason they were excluded. So a time limit has been introduced and also $N = 160$ suffered this problem. $N = 140$ seems to be a good compromise between our computational resources and the size of the problem.

4.1 CPLEX results

The following tables show the results obtained with *CPLEX*. Table 2 and 3 include how many instances exceeded the time limit. For that reason $N = 160$ and $N = 200$ will not be taken into consideration, since with the tabu search we could obtain a better solution and we would not be able to estimate how much far is the tabu search solution respect to the exact one. Furthermore, $N = 10$ and $N = 20$ will be also not taken into considerations since they seems not significant for time and complexity of the problem to the instances $N = 40$.

N	# exceeded	# instances
10	0/5	-
20	0/5	-
40	0/5	-
60	0/5	-
80	0/5	-
100	0/5	-
120	0/5	-
140	0/5	-
160	2/5	2,3
200	4/5	0,2,3,4

Table 2: Random instances exceeded

N	# exceeded	# instances
10	0/5	-
20	0/5	-
40	0/5	-
60	0/5	-
80	0/5	-
100	0/5	-
120	0/5	-
140	1/5	2
160	3/5	1,3,4
200	4/5	0,1,3,4

Table 3: Regularity instances exceeded

By looking at Table 4 and 5 , most solutions are the exact ones, but some of them are sub-optimal solutions obtained after 45 minutes of computation and they are marked with a * sign. Clearly the increasing of the size n of instances implies the increment of executions time.

N	Solution	Execution time(s)
pos_n40_0	264	0.472
pos_n40_1	250	0.825
pos_n40_2	252	0.837
pos_n40_3	242	0.912
pos_n40_4	258	0.737
pos_n60_0	470	1.824
pos_n60_1	444	6.686
pos_n60_2	488	6.841
pos_n60_3	418	2.720
pos_n60_4	486	15.877
pos_n80_0	692	42.740
pos_n80_1	652	22.881
pos_n80_2	692	13.576
pos_n80_3	684	53.081
pos_n80_4	726	46.019
pos_n100_0	948	112.628
pos_n100_1	908	166.058
pos_n100_2	952	194.950
pos_n100_3	1002	106.113
pos_n100_4	1028	120.726
pos_n120_0	1252	273.848
pos_n120_1	1270	400.508
pos_n120_2	1256	283.622
pos_n120_3	1226	240.301
pos_n120_4	1302	328.428
pos_n140_0	1528	411.273
pos_n140_1	1514	353.901
pos_n140_2	1558	270.911
pos_n140_3	1618	429.515
pos_n140_4	1572	426.417

Table 4: Random instances results

N	Solution	Execution time(s)
pos_n40_0	182	1.421
pos_n40_1	172	1.581
pos_n40_2	152	4.149
pos_n40_3	168	2.909
pos_n40_4	192	0.554
pos_n60_0	214	13.276
pos_n60_1	274	5.867
pos_n60_2	330	18.823
pos_n60_3	372	4.525
pos_n60_4	260	3.224
pos_n80_0	418	32.155
pos_n80_1	374	58.406
pos_n80_2	340	5.422
pos_n80_3	612	43.866
pos_n80_4	528	49.159
pos_n100_0	614	219.586
pos_n100_1	472	59.259
pos_n100_2	630	174.420
pos_n100_3	616	165.154
pos_n100_4	642	199.671
pos_n120_0	766	882.049
pos_n120_1	942	343.300
pos_n120_2	678	107.189
pos_n120_3	902	366.466
pos_n120_4	702	295.037
pos_n140_0	1096	831.651
pos_n140_1	856	2460.66
pos_n140_2	684*	2700.12*
pos_n140_3	950	1313.76
pos_n140_4	1054	492.785

Table 5: Regularity instances results

4.2 Tabu search calibration

The calibration of the paramenters are important in order to obtain a high quality solution in a decent amount of time. First we will calibrate the tabu length and then all the other parameters. In this case, we begin with only one stopping criteria, but we will consider whether it is necessary a combination of them or not. Since there are a lot of instances of different types, we will use 3 instances random and 3 regularity for each size n and we will run them 10 times each (except for $N = 140$), taking the average solution and execution time. The solutions with * remember that they have to be compared with a sub-optimal solution since in the *CPLEX* results time limit has been reached. Furthermore, we denote R (random) and S (shapes) as notation when we give the number of improving or worsening instances during the calibration (e.g. some improvements ($2R$ and $3S$) $\Rightarrow$ 2 random and 3 shapes instances are improved).

4.2.1 TSC1

In the first attempt, the following configuration of the parameters has been chosen:

- Tabu length: $\frac{n}{4}$
- Stopping criteria: *Max iterations*
- Max iterations: n
- Initial solution: *Nearest Neighbour*

N	Solution	Execution time(s)	Err. %
pos_n40_0	264	0.165	0.00
pos_n40_1	256	0.159	2.40
pos_n40_2	252	0.162	0.00
pos_n60_0	478	0.883	1.70
pos_n60_1	466	0.881	4.95
pos_n60_2	510	0.882	4.51
pos_n80_0	714	2.982	3.18
pos_n80_1	672	2.973	3.07
pos_n80_2	708	2.976	2.31
pos_n100_0	976	7.627	2.95
pos_n100_1	954	7.635	5.07
pos_n100_2	994	7.667	4.41
pos_n120_0	1338	16.837	6.87
pos_n120_1	1336	16.913	5.20
pos_n120_2	1314	16.789	4.62
pos_n140_0	1640	196.011	7.33
pos_n140_1	1600	196.111	5.68
pos_n140_2	1614	195.593	3.59

Table 6: Random instances TSC1

N	Solution	Execution time(s)	Err. %
pos_n40_0	182	0.162	0.00
pos_n40_1	176	0.160	2.33
pos_n40_2	152	0.160	0.00
pos_n60_0	220	0.874	2.80
pos_n60_1	284	0.875	3.65
pos_n60_2	332	0.872	0.61
pos_n80_0	428	2.946	2.39
pos_n80_1	376	2.940	0.53
pos_n80_2	340	2.985	0.00
pos_n100_0	618	7.749	0.65
pos_n100_1	474	7.604	0.42
pos_n100_2	634	7.653	0.63
pos_n120_0	808	16.831	5.48
pos_n120_1	1014	16.751	7.64
pos_n120_2	702	16.772	3.54
pos_n140_0	1184	195.810	8.03
pos_n140_1	878	195.837	2.57
pos_n140_2	692*	195.366*	1.17

Table 7: Regularity instances TSC1

4.2.2 TSC2

In this attempt, we tried to increase the tabu length, obtaining the following results:

- Tabu length: $\frac{n}{2}$
- Stopping criteria: *Max iterations*
- Max iterations: n
- Initial solution: *Nearest Neighbour*

N	Solution	Execution time(s)	Err. %
pos_n40_0	264	0.162	0.00
pos_n40_1	256	0.158	2.40
pos_n40_2	252	0.158	0.00
pos_n60_0	484	0.875	2.98
pos_n60_1	466	0.883	4.95
pos_n60_2	510	0.874	4.51
pos_n80_0	734	2.939	6.07
pos_n80_1	672	2.947	3.07
pos_n80_2	722	2.931	4.34
pos_n100_0	988	7.613	4.22
pos_n100_1	968	7.554	6.61
pos_n100_2	994	7.538	4.41
pos_n120_0	1352	16.685	7.99
pos_n120_1	1390	16.582	9.45
pos_n120_2	1314	16.654	4.62
pos_n140_0	1640	194.341	7.33
pos_n140_1	1682	194.115	11.10
pos_n140_2	1648	193.992	5.78

Table 8: Random instances TSC2

N	Solution	Execution time(s)	Err. %
pos_n40_0	182	0.159	0.00
pos_n40_1	176	0.160	2.33
pos_n40_2	152	0.166	0.00
pos_n60_0	218	0.868	1.87
pos_n60_1	286	0.872	4.38
pos_n60_2	332	0.873	0.61
pos_n80_0	428	2.931	2.39
pos_n80_1	376	2.932	0.53
pos_n80_2	340	2.941	0.00
pos_n100_0	618	7.535	0.65
pos_n100_1	474	7.519	0.42
pos_n100_2	634	7.557	0.63
pos_n120_0	800	16.482	4.44
pos_n120_1	1014	16.539	7.64
pos_n120_2	702	16.531	3.54
pos_n140_0	1192	192.507	8.76
pos_n140_1	910	192.613	6.31
pos_n140_2	696*	192.316*	1.75

Table 9: Regularity instances TSC2

As you can notice, it has been found more worsening than improvements, mostly on random generated instances, so for that reason the hypothesis of increasing the tabu list was discarded.

4.2.3 TSC3

In this attempt, we tried to decrease the tabu length, obtaining the following results:

- Tabu length: $\frac{n}{10}$
- Stopping criteria: *Max iterations*
- Max iterations: n
- Initial solution: *Nearest Neighbour*

N	Solution	Execution time(s)	Err. %
pos_n40_0	264	0.164	0.00
pos_n40_1	256	0.158	2.40
pos_n40_2	252	0.159	0.00
pos_n60_0	486	0.875	3.40
pos_n60_1	466	0.881	4.95
pos_n60_2	510	0.871	4.51
pos_n80_0	702	2.943	1.45
pos_n80_1	672	2.945	3.07
pos_n80_2	712	2.958	2.89
pos_n100_0	984	7.595	3.80
pos_n100_1	986	7.554	8.59
pos_n100_2	984	7.540	3.36
pos_n120_0	1338	16.659	6.87
pos_n120_1	1388	16.623	9.29
pos_n120_2	1312	16.536	4.46
pos_n140_0	1636	192.316	7.07
pos_n140_1	1666	192.774	10.04
pos_n140_2	1620	192.875	3.98

Table 10: Random instances TSC3

N	Solution	Execution time(s)	Err. %
pos_n40_0	184	0.159	1.10
pos_n40_1	176	0.159	2.33
pos_n40_2	152	0.159	0.00
pos_n60_0	218	0.874	1.87
pos_n60_1	280	0.871	2.19
pos_n60_2	332	0.869	0.61
pos_n80_0	428	2.914	2.39
pos_n80_1	376	2.932	0.53
pos_n80_2	340	2.935	0.00
pos_n100_0	618	7.537	0.65
pos_n100_1	472	7.501	0.00
pos_n100_2	634	7.522	0.63
pos_n120_0	810	16.468	5.74
pos_n120_1	1014	16.511	7.64
pos_n120_2	702	16.501	3.54
pos_n140_0	1180	191.988	7.66
pos_n140_1	904	192.119	5.61
pos_n140_2	692*	192.036*	1.17

Table 11: Regularity instances TSC3

From Table 10 and 11, it seems that results are improving in some instances ($4R$ and $4S$), but worsening in other ones ($7R$ and $3S$). Since on average improvements do not seem to be so significant, while worsening are more important, it might be more convenient to keep the tabu length of TSC1.

4.2.4 TSC4

After the calibration of the tabu length, since reducing the number of maximum iterations would give clearly worst outcomes, it has been tried increase them in order to see whether there are improvements or not, obtaining the following results:

- Tabu length: $\frac{n}{4}$
- Stopping criteria: *Max iterations*
- Max iterations: $\frac{3}{2}n$
- Initial solution: *Nearest Neighbour*

N	Solution	Execution time(s)	Err. %
pos_n40_0	264	0.249	0.00
pos_n40_1	256	0.237	2.40
pos_n40_2	252	0.242	0.00
pos_n60_0	478	1.320	1.70
pos_n60_1	466	1.316	4.95
pos_n60_2	510	1.406	4.51
pos_n80_0	698	4.652	0.87
pos_n80_1	664	4.705	1.84
pos_n80_2	708	4.671	2.31
pos_n100_0	968	12.193	2.11
pos_n100_1	954	11.849	5.07
pos_n100_2	984	11.147	3.36
pos_n120_0	1322	24.451	5.59
pos_n120_1	1336	24.656	5.20
pos_n120_2	1314	24.655	4.62
pos_n140_0	1632	288.544	6.81
pos_n140_1	1592	287.761	5.15
pos_n140_2	1614	287.208	3.59

Table 12: Random instances TSC4

N	Solution	Execution time(s)	Err. %
pos_n40_0	182	0.248	0.00
pos_n40_1	176	0.237	2.33
pos_n40_2	152	0.237	0.00
pos_n60_0	220	1.308	2.80
pos_n60_1	280	1.301	2.19
pos_n60_2	332	1.298	0.61
pos_n80_0	428	4.524	2.39
pos_n80_1	374	4.667	0.00
pos_n80_2	340	4.665	0.00
pos_n100_0	618	11.957	0.65
pos_n100_1	474	11.974	0.42
pos_n100_2	634	12.100	0.63
pos_n120_0	794	24.549	3.66
pos_n120_1	1014	24.408	7.64
pos_n120_2	702	24.507	3.54
pos_n140_0	1182	286.679	7.85
pos_n140_1	878	287.184	2.57
pos_n140_2	692*	286.203*	1.17

Table 13: Regularity instances TSC4

By looking at Table 12 and 13, there are good improvements ($7R$ and $4S$) on the solution of different instances, paying $\frac{3}{2}$ multiply factor compared to TSC1 on execution time with *maximum iteration* as stopping criteria. So changing to the parameters configuration from TSC1 to TSC4 is convenient.

4.2.5 TSC5

After the last results, it has been tried again to increase more the number of iterations in order to see whether the outcomes improve more or not, obtaining the following results:

- Tabu length: $\frac{n}{4}$
- Stopping criteria: *Max iterations*
- Max iterations: $2n$
- Initial solution: *Nearest Neighbour*

N	Solution	Execution time(s)	Err. %
pos_n40_0	264	0.303	0.00
pos_n40_1	256	0.295	2.40
pos_n40_2	252	0.297	0.00
pos_n60_0	478	1.666	1.70
pos_n60_1	466	1.662	4.95
pos_n60_2	506	1.676	3.69
pos_n80_0	698	5.659	0.87
pos_n80_1	664	5.683	1.84
pos_n80_2	708	5.657	2.31
pos_n100_0	968	14.672	2.11
pos_n100_1	942	14.636	3.74
pos_n100_2	984	14.624	3.36
pos_n120_0	1322	32.370	5.59
pos_n120_1	1336	32.214	5.20
pos_n120_2	1314	32.187	4.62
pos_n140_0	1630	379.588	6.68
pos_n140_1	1592	379.477	5.15
pos_n140_2	1614	379.064	3.59

Table 14: Random instances TSC5

N	Solution	Execution time(s)	Err. %
pos_n40_0	182	0.300	0.00
pos_n40_1	176	0.295	2.33
pos_n40_2	152	0.293	0.00
pos_n60_0	220	1.658	2.80
pos_n60_1	280	1.660	2.19
pos_n60_2	332	1.654	0.61
pos_n80_0	428	5.640	2.39
pos_n80_1	374	5.635	0.00
pos_n80_2	340	5.688	0.00
pos_n100_0	618	14.634	0.65
pos_n100_1	474	14.563	0.42
pos_n100_2	634	14.707	0.63
pos_n120_0	794	32.363	3.66
pos_n120_1	1014	32.426	7.64
pos_n120_2	702	32.147	3.54
pos_n140_0	1182	378.822	7.85
pos_n140_1	878	378.587	2.57
pos_n140_2	692*	377.682*	1.17

Table 15: Regularity instances TSC5

Also in this case there are some small improvements ($3R$) compared to TSC4. On the other side, it should be reasonable not to increase anymore the maximum number of iterations, since this would raise up the execution time and this could be a problem for larger N.

4.2.6 TSC6

After balancing the stopping criteria, it has been tried to change it with the maximum number of non-improving iterations, obtaining the following results:

- Tabu length: $\frac{n}{4}$
- Stopping criteria: *Max non-improving iterations*
- Max non-improving iterations: $\frac{n}{2}$
- Initial solution: *Nearest Neighbour*

N	Solution	Execution time(s)	Err. %
pos_n40_0	264	0.111	0.00
pos_n40_1	256	0.101	2.40
pos_n40_2	252	0.135	0.00
pos_n60_0	478	1.122	1.70
pos_n60_1	466	0.629	4.95
pos_n60_2	510	0.717	4.51
pos_n80_0	714	3.449	3.18
pos_n80_1	672	2.313	3.07
pos_n80_2	708	3.474	2.31
pos_n100_0	988	5.109	4.22
pos_n100_1	954	7.229	5.07
pos_n100_2	994	4.972	4.41
pos_n120_0	1338	21.005	6.87
pos_n120_1	1336	19.250	5.20
pos_n120_2	1314	13.200	4.62
pos_n140_0	1640	142.085	7.33
pos_n140_1	1592	291.150	5.15
pos_n140_2	1614	238.898	3.59

Table 16: Random instances TSC6

N	Solution	Execution time(s)	Err. %
pos_n40_0	182	0.140	0.00
pos_n40_1	176	0.105	2.33
pos_n40_2	152	0.106	0.00
pos_n60_0	220	0.730	2.80
pos_n60_1	286	0.622	4.38
pos_n60_2	332	0.715	0.61
pos_n80_0	428	3.337	2.39
pos_n80_1	376	1.920	0.53
pos_n80_2	340	2.158	0.00
pos_n100_0	618	6.225	0.65
pos_n100_1	474	5.672	0.42
pos_n100_2	634	5.495	0.63
pos_n120_0	794	28.566	3.66
pos_n120_1	1014	11.776	7.64
pos_n120_2	702	11.977	3.54
pos_n140_0	1184	233.574	8.03
pos_n140_1	878	267.547	2.57
pos_n140_2	696*	147.712*	1.75

Table 17: Regularity instances TSC6

The change on the stopping criteria gives better results in terms of execution time, but the quality of the solution worsen a lot ($8R$ and $4S$). Since reducing the number of maximum number of non-improving iterations would give worst outcomes, it has been increased the stopping criteria in order to see whether there are some improvements or not (results not shown), but no enhancements have been registered compared to TSC5.

4.2.7 TSC7

In this attempt, it has been tried to modify the initial solution in order to understand whether starting also from a possible worse initial solution choosen randomly could give some better outcomes. The results are shown in the following tables:

- Tabu length: $\frac{n}{4}$
- Stopping criteria: *Max iterations*
- Max iterations: $2n$
- Initial solution: *Random Init*

N	Solution	Execution time(s)	Err. %
pos_n40_0	265.6	0.280	0.61
pos_n40_1	260.4	0.272	4.16
pos_n40_2	256.6	0.273	1.83
pos_n60_0	488.4	1.585	3.91
pos_n60_1	463.6	1.582	4.41
pos_n60_2	508.8	1.589	4.26
pos_n80_0	716.4	5.428	3.53
pos_n80_1	673.2	5.436	3.25
pos_n80_2	722.0	5.440	4.34
pos_n100_0	986.4	14.203	4.05
pos_n100_1	948.0	14.250	4.41
pos_n100_2	989.4	14.215	3.93
pos_n120_0	1310.0	31.243	4.63
pos_n120_1	1320.0	31.192	3.94
pos_n120_2	1321.0	31.353	5.18
pos_n140_0	1650.8	374.637	8.04
pos_n140_1	1590.4	374.883	5.05
pos_n140_2	1670.0	375.387	7.19

Table 18: Random instances TSC7

N	Solution	Execution time(s)	Err. %
pos_n40_0	187.2	0.279	2.86
pos_n40_1	174.6	0.274	1.51
pos_n40_2	152.2	0.274	0.13
pos_n60_0	222.6	1.584	4.02
pos_n60_1	280.4	1.573	2.34
pos_n60_2	339.6	1.575	2.91
pos_n80_0	436.2	5.410	4.35
pos_n80_1	401.2	5.389	7.27
pos_n80_2	343.0	5.372	0.88
pos_n100_0	633.6	14.187	3.19
pos_n100_1	491.8	14.193	4.19
pos_n100_2	669.6	14.160	6.29
pos_n120_0	791.2	31.090	3.29
pos_n120_1	1017.8	31.134	8.05
pos_n120_2	716.2	31.308	5.63
pos_n140_0	1137.6	373.304	3.80
pos_n140_1	890.4	372.706	4.02
pos_n140_2	715.6*	371.985*	4.62

Table 19: Regularity instances TSC7

From Table 18 and 19, it is possible to see that the results are even worse than TSC6, so the stopping criteria seems not to be appropriate, hence for the moment it is convenient to keep TSC5 as calibration parameter combination.

4.2.8 TSC8

IN this attempt we retry to see whether what achieved in TSC2 does not depend from the initial solution. So it has been set the same tabu length of TSC2 and the maximum number of iteration found during the calibration ($2n$), changing the initial solution to *random init*. It has been obtained the following results:

- Tabu length: $\frac{n}{2}$
- Stopping criteria: *Max iterations*
- Max iterations: $2n$
- Initial solution: *Random Init*

N	Solution	Execution time(s)	Err. %
pos_n40_0	273.4	0.284	3.56
pos_n40_1	257.6	0.273	3.04
pos_n40_2	259.0	0.274	2.78
pos_n60_0	492.8	1.568	4.85
pos_n60_1	464.2	1.575	4.55
pos_n60_2	513.6	1.581	5.25
pos_n80_0	723.2	5.402	4.51
pos_n80_1	679.0	5.403	4.14
pos_n80_2	728.2	5.416	5.23
pos_n100_0	997.8	14.142	5.25
pos_n100_1	951.0	14.164	4.74
pos_n100_2	996.2	14.260	4.64
pos_n120_0	1356.2	31.579	8.32
pos_n120_1	1349.4	31.410	6.25
pos_n120_2	1319.4	31.485	5.05
pos_n140_0	1640.4	377.438	7.36
pos_n140_1	1606.0	377.335	6.08
pos_n140_2	1646.0	377.105	5.65

Table 20: Random instances TSC8

N	Solution	Execution time(s)	Err. %
pos_n40_0	187.6	0.274	3.08
pos_n40_1	174.0	0.272	1.16
pos_n40_2	152.8	0.274	0.53
pos_n60_0	223.2	1.581	4.30
pos_n60_1	289.4	1.581	5.62
pos_n60_2	346.0	1.576	4.85
pos_n80_0	433.0	5.447	3.59
pos_n80_1	405.8	5.398	8.50
pos_n80_2	341.2	5.400	0.35
pos_n100_0	645.0	14.068	5.05
pos_n100_1	485.0	14.243	2.75
pos_n100_2	666.6	14.123	5.81
pos_n120_0	796.2	31.627	3.94
pos_n120_1	1036.0	31.128	9.98
pos_n120_2	719.8	31.166	6.17
pos_n140_0	1164.4	373.507	6.24
pos_n140_1	896.8	374.715	4.77
pos_n140_2	712.4*	374.291*	4.15

Table 21: Regularity instances TSC8

The results are worse than TSC5, so this suggests to keep the calibration parameters of TSC5. It is important to underline that the results compared with TSC7 could confirm that the initial solution doesn't affect in any way the calibration of the tabu length and for this reason it is definitely discarded.

4.2.9 TSC9

Finally, we tried to change the stopping criteria with the number of maximum non-increasing iterations. Compared to TSC6, they have been increased, but on this occasion a time limit of 7 minutes has been added to shorten the execution. In order to perfectly evaluate this case in which we increased the number of non-improving iterations, it should not be used the time limit. The results are shown in the following tables:

- Tabu length: $\frac{n}{4}$
- Stopping criteria: *Max non-improving iterations + Time Limit*
- Max non-improving iterations: n
- Time Limit: 7 *minutes*
- Initial solution: *Random Init*

N	Solution	Execution time(s)	Err. %
pos_n40_0	268.6	0.302	1.74
pos_n40_1	258.8	0.298	3.52
pos_n40_2	254.0	0.308	0.79
pos_n60_0	491.8	2.113	4.64
pos_n60_1	461.8	2.029	4.01
pos_n60_2	500.8	2.330	2.62
pos_n80_0	715.8	6.876	3.44
pos_n80_1	677.6	7.077	3.93
pos_n80_2	719.2	7.161	3.93
pos_n100_0	982.6	18.712	3.65
pos_n100_1	951.0	21.270	4.74
pos_n100_2	1007.2	17.284	5.80
pos_n120_0	1328.2	41.840	6.09
pos_n120_1	1319.4	41.950	3.89
pos_n120_2	1329.0	42.842	5.81
pos_n140_0	1644.0	416.442	7.59
pos_n140_1	1600.8	420.498	5.73
pos_n140_2	1653.2	420.803	6.11

Table 22: Random instances TSC9

N	Solution	Execution time(s)	Err. %
pos_n40_0	184.2	0.310	1.21
pos_n40_1	176.0	0.281	2.33
pos_n40_2	152.0	0.271	0.00
pos_n60_0	221.6	1.976	3.55
pos_n60_1	281.6	1.976	2.77
pos_n60_2	340.4	2.210	3.15
pos_n80_0	425.8	6.344	1.87
pos_n80_1	395.4	7.388	5.72
pos_n80_2	343.2	6.693	0.94
pos_n100_0	630.4	17.323	2.67
pos_n100_1	503.8	19.534	6.74
pos_n100_2	664.8	15.331	5.52
pos_n120_0	792.8	40.943	3.50
pos_n120_1	1015.2	48.060	7.77
pos_n120_2	707.4	45.471	4.34
pos_n140_0	1130.0	420.724	3.10
pos_n140_1	890.4	420.663	4.02
pos_n140_2	716.4*	413.234*	4.74

Table 23: Regularity instances TSC9

It is possible to see that most of the execution of instances for $N = 140$ are probably finished with a time limit, but in general it seems there is no evidence for improvements for all other instances compared to TSC5.

4.3 Tabu search results

We have seen tep-by-step how TSC5 seems to be the best calibration for configuration parameters of the tabu search. In Table 24 and 25 there are the final results for all the instances we have solved with *CPLEX* using TSC5 calibration.

We can see how most of the percentage errors are under 5% ($12R$ and $12S$) which could be considered a good result. On the other side, there are some instances ($6R$ and $5S$) with a percentage error between 5% and 8% which is acceptable but it could be improved and only one instance has more than 8% of error. In general, we can see that all the instances are solved faster than the ones *CPLEX*.

Finally, we can state that the average percentage error for the random generated instances is 4.72% while for the shape instances is 4.66%, noticing that there is not so much difference between the types of instances. Hence the average percentage error of the method is about 4.69%.

N	Solution	Execution time(s)	Err. %
pos_n40_0	264	0.303	0.00
pos_n40_1	256	0.295	2.40
pos_n40_2	252	0.297	0.00
pos_n40_3	244	0.302	0.83
pos_n40_4	274	0.299	6.20
pos_n60_0	478	1.666	1.70
pos_n60_1	466	1.662	4.95
pos_n60_2	506	1.676	3.69
pos_n60_3	428	1.658	2.39
pos_n60_4	496	1.668	2.06
pos_n80_0	698	5.659	0.87
pos_n80_1	664	5.683	1.84
pos_n80_2	708	5.657	2.31
pos_n80_3	692	5.710	1.17
pos_n80_4	740	5.657	1.93
pos_n100_0	968	14.672	2.11
pos_n100_1	942	14.636	3.74
pos_n100_2	984	14.624	3.36
pos_n100_3	1008	14.622	0.60
pos_n100_4	1040	14.783	1.17
pos_n120_0	1322	32.370	5.59
pos_n120_1	1336	32.214	5.20
pos_n120_2	1314	32.187	4.62
pos_n120_3	1258	32.372	2.61
pos_n120_4	1344	32.562	3.23
pos_n140_0	1630	379.588	6.68
pos_n140_1	1592	379.477	5.15
pos_n140_2	1614	379.064	3.59
pos_n140_3	1686	384.935	4.20
pos_n140_4	1584	385.866	0.76

Table 24: Random instances final results

N	Solution	Execution time(s)	Err. %
pos_n40_0	182	0.300	0.00
pos_n40_1	176	0.295	2.33
pos_n40_2	152	0.293	0.00
pos_n40_3	168	0.299	0.00
pos_n40_4	192	0.302	0.00
pos_n60_0	220	1.658	2.80
pos_n60_1	280	1.660	2.19
pos_n60_2	332	1.654	0.61
pos_n60_3	372	1.659	0.00
pos_n60_4	270	1.654	3.85
pos_n80_0	428	5.640	2.39
pos_n80_1	374	5.635	0.00
pos_n80_2	340	5.688	0.00
pos_n80_3	638	5.655	4.25
pos_n80_4	574	5.666	8.71
pos_n100_0	618	14.634	0.65
pos_n100_1	474	14.563	0.42
pos_n100_2	634	14.707	0.63
pos_n100_3	642	14.744	4.22
pos_n100_4	660	14.744	2.80
pos_n120_0	794	32.363	3.66
pos_n120_1	1014	32.426	7.64
pos_n120_2	702	32.147	3.54
pos_n120_3	918	32.295	1.77
pos_n120_4	752	32.299	7.12
pos_n140_0	1182	378.822	7.85
pos_n140_1	878	378.587	2.57
pos_n140_2	692*	377.682*	1.17
pos_n140_3	1016	380.666	6.95
pos_n140_4	1116	381.187	5.88

Table 25: Regularity instances final results

4.4 Conclusions

In this report we have seen some implementation details of both *CPLEX* and tabu search.
Results for both methods have been presented and we have seen that *CPLEX* clearly performs better in terms of solution quality, since it is an exact method, while tabu search is preferable in terms of execution time, dropping something in solution quality. In a real life scenario, the company should use *CPLEX* or tabu search depending on the time constraints and the size of the instance they are going to create. Obviously, the larger the instance, the more appropriate is the use of tabu search, even if it gives a lower quality solution, since the exact method would take too much time.
From a technical point of view, a refinement of the tabu search parameters could be useful to try to improve the performances, especially in terms of the quality of the solution. More tests and more runs should be done in order to effectively understand where to improve the parameters calibration. In addition, more optimizations could be done on the initial solution, trying also to implement different local search algorithms to diversify the search, e.g. *3-OPT*, *Christofides*, *etc.* In fact, it is not only important to start with a good solution, but also to start in a good position in the solution space. In this case, random multi-start might be suitable to achieve diversification, but it would take more time to run.

A Instructions

In this section there are instruction useful to correctly create a representation of the instance, run it and see the results in a txt file. All the results will be stored in the test directory positioned in both lab_ex_1 and lab_ex_2.
There will be three instances for testing:

- pos_n80_random.dat = pos_n80_0.dat random generated
- pos_n80_regularity.dat = pos_n80_0.dat shapes generated
- pos_n60_regularity.dat = pos_n60_3.dat shapes generated

A.1 Draw the instance

From the lab_ex_1 or lab_ex_2 directory, do the following.
Commands (the regularity instance is analogous by substituting "random" with "regularity" and also $N = 60$ is similar):

- python3 plot.py <number of points> random

A.2 Solve with CPLEX

From the lab_ex_1 directory, do the following.
Commands (the regularity instance is analogous by substituting "random" with "regularity" and also $N = 60$ is similar):

- make clean
- make
- ./main x.dat test/savedCosts/savedCosts_n80_random.dat x ReadCosts >> test/results /results_n80_random.txt

If you need to check how an instance and related fixed costs are generated for random instances:

./main test/posFiles/namefile.dat test/savedCosts/namefile.dat <number of points> x random

For shapes:

./main test/posFiles/namefile.dat test/savedCosts/namefile.dat <number of points>

The generation feature is only available with lab_ex_1

A.3 Solve with Tabu Search

From the lab_ex_2 directory, do the following.
Commands (the regularity instance is analogous by substituting "random" with "regularity" and also $N = 60$ is similar):

- make clean
- make
- ./main test/savedCosts/savedCosts_n80_random.dat <tabu_length> <max_it> <init> >> test/results/results_n80_random_<init>.txt
  where $init \in \{0, 1\}$. If $init = 0$, then $randomInit$ else $NearestNeighInit$
- Suggestion: $tabu\_length = 20$ and $max\_it = 160$

If you need to change the stopping criteria, modify accordingly the variables in *main.cpp* and the commented code in the *solve()* method in *TSPsolver.cpp*.
Furthermore, if you need to see the solution, modify the commented code in *main.cpp*.