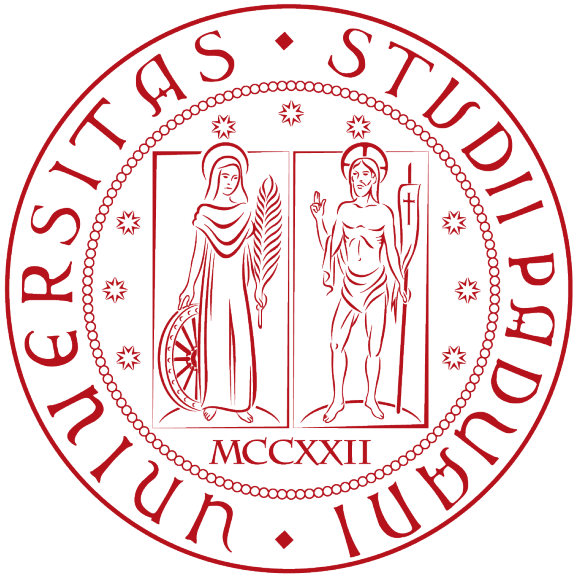
Università degli Studi di Padova

Master Degree in Computer Science



Methods and Models for Combinatorial Optimization

Lab exercise report

Lorenzo Ceccon - 1154186

2020/2021

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PAGE

LIST OF ALGORITHMS

1 Introduction

The proposed exercise is divided into two parts; the first part consists in implementing an Integer Linear Programming model using the CPLEX APIs. The second part consists, instead, of developing a meta-heuristic for the assigned problem and compare the performances obtained by this algorithm with those obtained using CPLEX.

The combinatorial optimization problem assigned is to find the shortest path that a drill has to do over a board in order to minimize the time necessary to make the holes on it. Basically it is about finding the minimum weight hamiltonian cycle, that is a Travelling Salesman Problem (TSP).

This report aims to present the design choices implemented and analyze the results by comparing the performance of the solutions obtained through CPLEX with those obtained through the use of meta-heuristics.

2 First part - CPLEX

The Integer Linear Programming (ILP) model has been implemented as proposed in the assignment.
The classes involved in this part are as follows:

- *problem.**: it's the class that creates the problem starting from a text file containing the matrix of the costs (times) that the drill takes to go from one hole to another. It also takes care of saving the minimum cost and the solution of the problem;
- *cplex/cplex_solver.**: it contains the implementation of the proposed model and uses the APIs provided by the CPLEX library. The details of this class will be reported later.

2.1 Model implementation

First of all, the decision variables were defined as indicated by the model and reported here below:

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

In addition to the index, has been assigned a name to the variables containing the row and column indexes in which they are located in order to make them easier to read and reference. Furthermore, for both matrices x and y, the variables that refer to the main diagonal of the matrix $(i = j)$ haven't been created as arcs of this type are not allowed.

```
int current_y_var_position = 0;
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        if (i == j) continue;
        char y_type = 'B';
        double obj = costs[i][j];
        double lb = 0.0;
        double ub = 1.0;
        snprintf(name, NAME_SIZE, "y_%d_%d", indexes[i], indexes[j]);
        char* y_name = (char*) &name[0];
        CHECKED_CPX_CALL(CPXnewcols, env, lp, 1, &obj, &lb, &ub, &y_type,
                &y_name);
        map_y[i][j] = current_y_var_position;
        ++current_y_var_position;
```

```
    }
}

const int x_init = CPXgetnumcols(env, lp);
int current_x_var_position = x_init;

for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        if (i == j) continue;
        char x_type = 'I';
        double obj = 0.0;
        double lb = 0.0;
        double ub = CPX_INFBOUND;
        snprintf(name, NAME_SIZE, "x_%d_%d", indexes[i], indexes[j]);
        char* x_name = (char*) &name[0];
        CHECKED_CPX_CALL(CPXnewcols, env, lp, 1, &obj, &lb, &ub, &x_type,
                &x_name);
        map_x[i][j] = current_x_var_position;
        ++current_x_var_position;
    }
}
```

Algorithm 2.1. Implementation of decision variables.

The constraints are implemented as specified in the paper and reported here:

$$
\begin{array}{lll}
1) & \sum_{j:(i,j)\in A} y_{\mathrm{ij}} = 1 & \forall i \in N \\
2) & \sum_{i:(i,j)\in A} y_{\mathrm{ij}} = 1 & \forall j \in N \\
3) & \sum_{i:(i,k)\in A} x_{\mathrm{ik}} - \sum_{j:(k,j),j\neq 0} x_{\mathrm{kj}} = 1 & \forall k \in N \backslash \{0\} \\
4) & x_{\mathrm{ij}} \leq (|N|-1) y_{\mathrm{ij}} & \forall (i,j) \in A, j \neq 0
\end{array}
$$

Not focusing on the definition of the constraints relating to the incoming and outcoming flow (constraint 1 and 2 here above) that are easy to understand and implement; regarding the third group of constraints, which concerns the difference in the amount of incoming and outcoming flow for each node $k \in N$, has been optimized by avoiding the insertion of the variables x_{ik} and x_{kj} with $k=0$, $j=0$, $k=i$ and $k=j$.

```
for (int k = 0; k < N; k++) {
    if (k == first_index) continue;
    std::vector<int> idx;
    std::vector<double> coef;
    for (int i = 0; i < N; i++) {
        if (k == i) continue;
```

```
        idx.push_back(map_x[i][k]);
        coef.push_back(1);
    }
    for (int j = 1; j < N; j++) {
        if (k == j) continue;
        idx.push_back(map_x[k][j]);
        coef.push_back(-1);
    }
    char sense = 'E';
    double rhs = 1.0;
    snprintf(name, NAME_SIZE, "flow_diff_%d", k);
    char* cname = (char*)(&name[0]);
    int matbeg = 0;
    CHECKED_CPX_CALL(CPXaddrows, env, lp, 0, 1, idx.size(), &rhs,
            &sense, &matbeg, &idx[0], &coef[0], NULL, &cname);
}
```

Algorithm 2.2. Implementation of the third group of constraints (difference of amount of flows).

2.2 Test results

The results of the tests performed by the implemented model will be reported here below. The first table (Table 2.1) will show the average size that CPLEX can solve in a given amount of time. In order to have a fairly precise size, the tests were performed on ten different instances and the average value obtained was recorded.

Execution time	Size
0,1s	10
1s	25
10s	45
60s	75
300s	100

Table 2.1. Average size that CPLEX is able to solve in a given amount of time n.

The following table (Table 2.2), instead, will show the results obtained running the model developed on 50 instances (5 instances for each size between 10 and 100). In addition to the optimal solution, will be reported the minimum, maximum and average times for solving the instances of a certain size N.
The graph (Figure 2.1) below will show the exponential growth of the time required by CPLEX to solve the problem at the increasing of the size.

	Solutions							
Size	**1**	**2**	**3**	**4**	**5**	**Min time**	**Max time**	**Avg time**
10	24,5274	28,4138	33,0114	31,9688	36,0208	0,07	0,16	0,09
20	80,1507	92,5091	84,8332	87,9096	76,0732	0,30	0,59	0,45
30	151,242	153,796	139,384	137,578	129,035	0,74	4,32	1,67
40	209,251	207,889	194,012	208,048	226,795	3,24	7,22	5,40
50	279,951	303,48	304,665	313,753	289,54	12,29	16,40	13,81
60	377,513	379,438	344,142	373,352	368,585	21,93	34,58	25,86
70	482,742	455,29	505,734	439,449	464,861	34,41	58,39	44,17
80	593,948	555,221	555,303	601,091	573,348	62,74	126,25	94,25
90	675,167	665,617	656,7	695,764	666,423	81,57	181,44	133,66
100	746,278	747,508	780,888	761,164	793,127	136,16	513,51	297,10

Table 2.2. Optimal solutions for each instance tested and the relatives minimum, maximum and average times calculated to solve the instances.

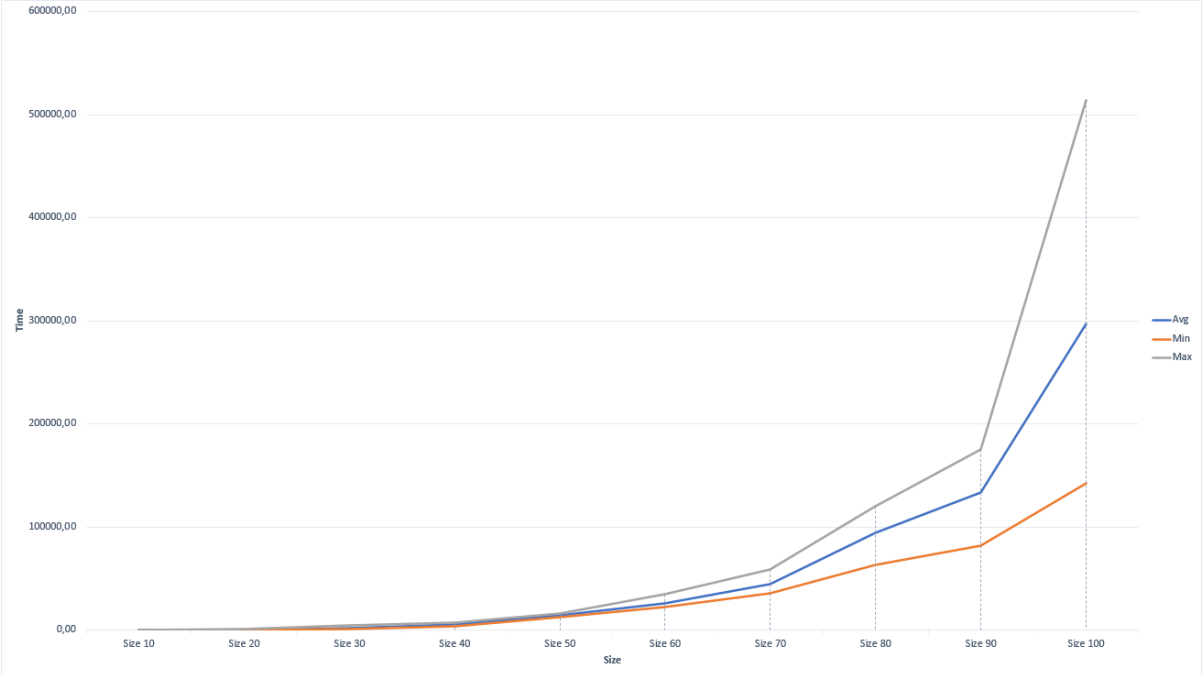


Figure 2.1. This is a graph that show the minimum, maximum and average time taken by CPLEX to solve the TSP at the increasing size.

3 Second part - Metaheuristic

For this part of the project, it was decided to implement the Ant Colony Optimization (ACO) algorithm as an ad-hoc meta-heuristic. After that it was decided to try to improve the algorithm by integrating Simulated Annealing (SA) to it. For simplicity, this modified version will be presented with the label $ACO+SA$ in the following sections of this paper.
The classes involved in this part are as follows:

- *problem.**: as for first part;
- *ant_colony/ant_colony_solver.**: it contains the implementation of the ACO algorithm with the SA improvement.

3.1 Analysis of the ACO

It was chosen to develop the first Ant Colony Optimization algorithm developed by Marco Dorigo and illustrated very simply here: Ant Colony Optimization Algorithm.
The algorithm can be summarized as following:

- at each iteration, a group of ants are randomly placed in the nodes;
- a path is generated for each ant through the use of a probability matrix that takes into account the distance of the nodes not yet visited and the amount of pheromone released by the ants in the previous iterations (at the first iteration the pheromone matrix is composed of only 1). Arcs crossed on the previous iterations or with lower costs will therefore be more likely to be chosen;
- if an ant finds a better solution of the current one, it is saved as a best solution;
- at the end of each iteration, each ant will spread a quantity of pheromone for each edge visited and part of the pheromone deposited in the previous iterations will be evaporated.

The probability matrix is calculated using the following formula:

$$p_{xy}^{k} = \frac{(\tau_{xy}^{\alpha})(\eta_{xy}^{\beta})}{\sum_{z \in allowed_x}(\tau_{xz}^{\alpha})(\eta_{xz}^{\beta})}$$

where τ_{xy} is the quantity of pheromone present between the nodes x and y, η_{xy} is the visibility, calculated as the inverse of the distance between x and y ($1/d$), finally, α and β, are arbitrary values that, respectively, regulate the importance of the pheromone matrix and the visibility matrix.

```
int start_city = rand() % num_cities;
ants_paths[j][0] = start_city;
allow_list.erase(allow_list.begin() + ants_paths[j][0]);
int count = 0;
probability_matrix.resize(num_cities);
while (allow_list.size() > 0) {
    double probability_sum = 0.0;
    probability_matrix[ants_paths[j][count]].resize(num_cities);

    for (int k = 0; k < allow_list.size(); ++k) {
        probability_matrix[ants_paths[j][count]][allow_list[k]] =
            pow(pheromones_matrix[ants_paths[j][count]][allow_list[k]], alpha) *
            pow(visibility_matrix[ants_paths[j][count]][allow_list[k]], beta);
        probability_sum+=probability_matrix[ants_paths[j][count]][allow_list[k]];
    }
    for (int k = 0; k < allow_list.size(); ++k) {
        probability_matrix[ants_paths[j][count]][allow_list[k]] =
            probability_matrix[ants_paths[j][count]][allow_list[k]] /
            probability_sum;
    }

    double random = ((double) rand() / (RAND_MAX));
    double prob_tot = 0.0;
    int next_city = -1;
    int element_to_remove = -1;
    for (int k = 0; k < allow_list.size(); ++k) {
        prob_tot += probability_matrix[ants_paths[j][count]][allow_list[k]];
        if (random <= prob_tot) {
            next_city = allow_list[k];
            element_to_remove = k;
            break;
        }
    }
    count++;
    ants_paths[j][count] = next_city;
    allow_list.erase(allow_list.begin() + element_to_remove);
}
```

Algorithm 3.1. Calculation of the probability matrix and creation of the path of an ant.

The calculation for updating the pheromone is done through the use of the following formula:

$$\tau_{xy} = (1 - \rho)\tau_{xy} + \sum_{k}^{m} \Delta\tau_{xy}^{k}$$

where ρ is an arbitrary value that represent the pheromone coefficient, m is the number of ants and $\Delta\tau_{xy}^{k}$ is the pheromone spread by the ant k and it's equal to:

$$\Delta\tau_{xy}^{k} = \begin{cases} Q/L_k & \text{if the ant k go across the arc from x to y} \\ 0 & \text{otherwise} \end{cases}$$

where Q is a constant value and L_k is the length of path.

```
for (int j = 0; j < num_cities; j++) {
    delta_tau_table[j].resize(num_cities);
    for (int k = 0; k < num_cities; k++) {
        delta_tau_table[j][k] = 0;
    }
}
for (int j = 0; j < ants_paths.size(); j++) {
    for (int k = 0; k < ants_paths[j].size() - 1; k++) {
        delta_tau_table[ants_paths[j][k]][ants_paths[j][k+1]] +=
        this->q / costs[ants_paths[j][k]][ants_paths[j][k + 1]];
    }
    delta_tau_table[ants_paths[j][num_cities - 1]][ants_paths[j][0]] +=
    this->q / costs[ants_paths[j][ants_paths[j].size() - 1]][ants_paths[j][0]];
}

for (int j = 0; j < num_cities; j++) {
    for (int k = 0; k < num_cities; k++) {
        pheromones_matrix[j][k] = (1 - rho) *
        pheromones_matrix[j][k] + delta_tau_table[j][k];
    }
}
```

Algorithm 3.2. Updating of the pheromone matrix.

3.2 Analysis of the Simulated Annealing integration

To try to improve the ACO algorithm it was decided to try to integrate Simulated Annealing to it.

Before describing the algorithm, it's a good thing define what the parameter

called *temperature* represents. The algorithm uses it to calculate the probability with which to accept even worse solutions. Before the execution of the algorithm it is setted to a certain maximum temperature and at each iteration it decreases until it reaches a minimum temperature and the algorithm stops.

The algorithm is very simple and it was applied to the ACO algorithm as follows:

- at the end of each ACO iteration the best path of the generation is saved to apply the simulated annealing to it;
- if the temperature left is greater than the minimum temperature the algorithm can be executed;
- at each iteration the SA pick a random neighbour doing the swap of two node;
- the path of the ant is replaced by that calculated by the SA if the cost of the new path is lower, otherwise it can be accepted with a certain probability depending on the current temperature and the difference between the two costs calculated as below:

  $$e^{(-\frac{c^{SA}-c^{OLD}}{T})}$$

  where T is the current temperature, c^{SA} is the cost of the new solution found by the SA and c^{OLD} is the cost of the current solution;
- the temperature is updated as follow:

  $$T = T * T_{max}$$

  where T_{max} is the maximum temperature.

3.3 Test results

Since a Simulated Annealing algorithm has been developed as a modification to Ant Colony Optimization, we have chosen to export this algorithm to an external class and adapt it in order to compare the ACO and the ACO+SA not only with the model built with CPLEX but also with a simple heuristic like the SA. This class can be found into the folder *simulated_annealing*.

Before analyzing the results obtained, here below are reported find the configurations used for the tests of all three algorithms.

- Simulated Annealing (SA):
  - max temperature = 0.995;
  - min temperature = 1e-09;
  - number of iterations = number of nodes * 20.
- Ant Colony Optimization (ACO):
  - number of ants = number of nodes of the problem;
  - $\alpha = 1.0$;
  - $\beta = 2.0$;
  - $\rho = 0.1$;
  - q = 1.0;
  - max iterations = number of nodes * 8.
- Ant Colony Optimization with Simulated Annealing (ACO+SA):
  - number of ants = number of nodes of the problem;
  - $\alpha = 1.0$;
  - $\beta = 2.0$;
  - $\rho = 0.1$;
  - q = 1.0;
  - max iterations = number of nodes of the problem * 8;
  - max temperature = 0.995;
  - min temperature = 1e-08;
  - number of iterations = number of nodes of the problem * 4.

The tests were executed on the same instances used for the first part of the project. Each instance it was performed 7 times. For the tables Table 3.1, Table 3.2 and Table 3.3 no time limits have been set.

At the end of this section, a chart (Figure 3.1) where it is possible to see the average gaps between the results obtained by the three heuristics with respect to the optimal solution is shown.

SIMULATED ANNEALING									
	Avg Results								
Size	**1**	**2**	**3**	**4**	**5**	**Avg time**	**Min gap**	**Max gap**	**Avg gap**
10	24,5274	28,4138	33,0114	31,9688	36,0208	0,56	0,00	0,00	0,00
20	80,1507	95,9101	90,9343	87,9096	77,0199	1,34	0,00	7,19	2,14
30	158,5967	162,8627	171,8160	139,1940	140,1570	2,35	1,17	23,27	8,76
40	241,5617	219,6323	221,9523	253,7513	245,9397	3,54	5,65	21,97	13,18
50	325,7787	340,3650	340,1383	365,3993	328,0213	4,98	11,64	16,46	13,98
60	419,7067	464,1260	384,9317	398,5510	436,9133	6,79	6,75	22,32	14,13
70	555,1013	507,1987	576,7740	473,8233	557,6727	11,64	7,82	19,97	13,64
80	679,0897	630,2643	650,6407	682,0587	654,4727	14,81	13,47	17,17	14,53
90	836,5000	806,6347	739,7903	827,2803	774,4607	17,34	12,65	23,90	18,57
100	910,1263	798,5903	928,1800	875,9850	971,2423	20,96	6,83	22,46	17,04

Table 3.1. Results obtained using the SA algorithm. For each size is reported the average results obtained, the average time of execution and the minimum, maximum e average gap obtained from the optimal solution.

It can be observed that this algorithm does not always guarantee good results. Sometimes it's possible to obtain good results (gaps around 7-8%) compared to the optimal solution obtained with CPLEX but in most cases results obtained have gaps greater than 20% which are not really acceptable. While for large problems the average gaps are quite high, for small instances (number of nodes less or equal than 30) the execution times are higher than CPLEX.

A possible (untested) solution could be to modify the mutation algorithm randomizing between the swap of two nodes and the reverse of a subsequence of nodes instead of using only the swap technique.

ANT COLONY OPTIMIZATION									
	Avg Results								
Size	**1**	**2**	**3**	**4**	**5**	**Avg time**	**Min gap**	**Max gap**	**Avg gap**
10	24,5274	28,4138	33,0114	31,9688	36,0208	0,02	0,00	0,00	0,00
20	80,1507	92,6884	85,9798	87,9096	76,0732	0,12	0,00	1,85	0,37
30	152,4270	155,9867	144,1387	137,7927	129,4370	0,56	0,18	1,78	0,76
40	213,1913	207,9290	195,1790	210,1393	231,8300	1,75	0,00	3,02	1,43
50	285,7867	310,4837	323,3520	325,3097	296,5030	4,30	2,27	6,72	3,89
60	396,1790	385,3027	354,2603	378,4600	375,3720	8,57	1,45	4,49	2,51
70	508,2037	466,3300	541,9060	453,7517	490,4653	15,67	2,15	7,54	4,81
80	616,0483	584,6577	579,3467	625,1413	599,1557	26,59	3,76	5,83	4,60
90	728,5707	702,8407	695,0967	768,0107	697,9890	41,96	4,81	10,02	7,17
100	792,6873	774,2997	800,0877	795,3063	814,0567	63,62	1,72	5,75	3,55

Table 3.2. Results obtained using the ACO algorithm. For each size is reported the average results obtained, the average time of execution and the minimum, maximum e average gap obtained from the optimal solution.

With the use of ant colony optimization is possible to notice that the data collected are much better. In most cases gaps of less than 5% have been obtained, only in some cases gaps of around 10% have been obtained but the average gaps are nevertheless good. Execution times are also excellent, ranging from 0.02 seconds to resolve instances with 10 nodes to 63.62 seconds to resolve instances with 100 nodes.

ANT COLONY OPTIMIZATION WITH SIMULATED ANNEALING									
	Avg Results								
Size	**1**	**2**	**3**	**4**	**5**	**Avg time**	**Min gap**	**Max gap**	**Avg gap**
10	24,5274	28,4138	33,0114	31,9688	36,0208	0,02	0,00	0,00	0,00
20	80,1507	92,5091	85,0111	87,9096	76,0732	0,13	0,00	0,21	0,04
30	151,3923	156,3740	140,8197	137,6853	129,0353	0,59	0,00	1,68	0,58
40	213,2400	207,8890	194,3947	209,6970	228,3220	1,77	0,00	1,91	0,71
50	287,3437	306,6790	319,4507	327,0370	292,5583	4,28	1,04	4,85	2,76
60	395,4013	383,6617	350,3127	375,8797	375,1950	8,49	0,68	4,74	2,02
70	505,2663	464,7197	532,9313	449,7837	491,0290	15,55	2,07	5,63	4,02
80	617,8813	578,8917	577,3357	621,7983	596,5557	26,43	3,44	4,26	3,95
90	724,8823	698,5577	694,8373	751,7777	695,2390	42,08	4,32	8,05	6,10
100	787,3340	770,1750	794,3093	794,1677	815,7697	63,61	1,72	5,50	3,49

Table 3.3. Results obtained using the ACO+SA algorithm. For each size is reported the average results obtained, the average time of execution and the minimum, maximum e average gap obtained from the optimal solution.

With the integration of Simulated Annealing in the algorithm it's possible to see that the results are even better than the previous ones, obtaining on average an improvement of the gaps for all the dimensions of the analyzed instances. Compared to the previous solution, gap reductions ranging from 0.06% to 1.13%.

The execution times are also almost the same as the Simulated Annealing is executed only on the best solution found by the ACO on each generation. Perhaps better results could be achieved by running SA on multiple solutions from the same generation.

The table here below (Table 3.4) shows the results obtained running both the ACO and ACO+SA algorithms with and without time limits; it's easy to noticed that by reducing execution times by 2-3 times, the gaps get worse on average only by 0.57%.

Size	**Time**	**Avg ACO gap**	**Avg ACO+SA gap**
70	Unlimited (15,6s)	4,81	4,02
	5s	4,88	4,15
80	Unlimited (26,5s)	4,60	3,95
	10s	5,62	4,35
90	Unlimited (41,9s)	7,17	6,10
	20s	7,32	7,36
100	Unlimited (63,6s)	3,55	3,49
	30s	4,50	4,07

Table 3.4. Gaps obtained running the ACO and the ACO+SA on large instances without limit time and with stringent time limits.

In conclusion, it can be said that the Ant Colony Optimization technique integrated with the use of Simulated Annealing turns out to be a good heuristic for the TSP. Even using rather stringent time limits it is possible to obtain solutions that are very close to the optimal solution.

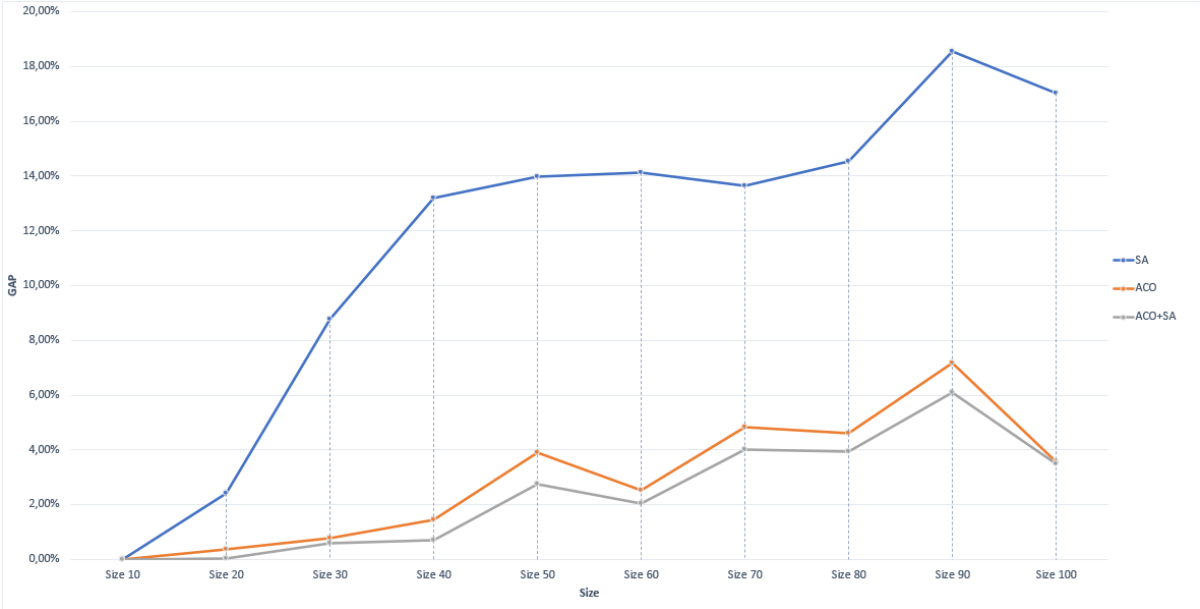


Figure 3.1. This graph shows the average gap between the optimal solution found with CPLEX and the best solutions found using the three algorithms (SA, ACO and ACO+SA).

4 Info

4.1 Creation of test instances

In order to create some instances for testing the software, in the root of the project there is a python script named *generate_random_instances.py* that create and save a text file inside the *inputs* folder. To generate an instance of size N just execute the following statement:

python generate_random_distances.py N

Otherwise, it's necessary to create a text file (using the .txt format) where in the first line is reported the dimension N of the problem and in the following lines the matrix of the distances.

4.2 Compilation and execution instructions

In order to compile the code is necessary to run the command *make* from the root of the project. In case of errors it's necessary to run the command *make clean* in order to clean the project and, then, retry to execute the command *make*.

To execute the software is necessary to run the *./main.out* command in the terminal. The user will be asked which algorithm he want to run between CPLEX, Ant Colony Optimization and Simulated Annealing. If Ant Colony Optimization is chosen, you will be asked if you want the ACO or ACO+SA version. Then you will be asked if you want to enter a time limit (pressing enter will set 5 minutes automatically). Finally, you will be asked on which instance you want to run the chosen algorithm, you must then type the name of a text file present inside the *inputs* folder or just press enter if you want to run the code on all the instances present in that folder.

4.3 Configuration used

The code was written using a Macbook Pro with the following configuration:

- macOS Big Sur version 11.2.2
- g++ Apple clang version 12.0.0
- CPLEX version 12.10

The software, in addition to being tested in the configuration just described, was also tested via ssh by connecting to the computers of the laboratory (labTA) that has the following configuration:

- Ubuntu 18.04.5 LTS
- g++ version 7.5.0
- CPLEX version 12.8

In both configurations the software compiled and ran correctly. To test the program in other configurations, just edit the *CPX_BASE* and *CPX_LIBDIR* entries in the *Makefile*.