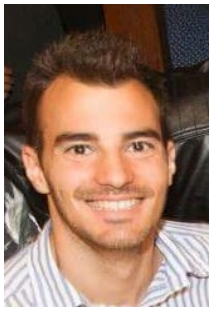
Università degli Studi di Padova

Wind Farm Cable Problem

Advanced combinatorial optimization algorithms
Ricerca Operativa 2



Piona Davide
1149616

Benini Michele
1139089

Academic Year 2017-2018

CONTENTS

1 DEVELOPMENT ENVIRONMENT

To allow replicability of our experiments, we have added some information about the system and the environment that we used for the following research. We used the following development environment:

- CPLEX : version 0.1-2018
- Linux Ubuntu : version 18.04 LTS (Operative System)
- C (language)
- Sublime text (IDE / text editor)
- Github (web-based version control using Git)
- Gnuplot (library plot)

1.1 REAL WORLD INFO

As in Martina Fischetti and Pisinger, 2018 research, we used a library of real-life instances that are publicly available for benchmarking. For benchmark purposes, we collected the data of five real wind farms in operation in United Kingdom (wf02, wf03, wf05) and Denmark (wf01, wf04). Different types of cable with different costs, capacities and electrical resistances are available on the market; we considered 5 sets of real cables, named cb01, cb02, cb03, cb04, and cb05. The turbines in each wind park are of the same type, so we can express the cable capacities as the maximum number of turbines that it can support. Finally, we list the maximum number of connections to the substation, or to be specific, the input parameter C. The **??** table contains all the information.

name	site	turbine type	no. of turbines	C	allowed cables
wf01	Horn Rev 1	Vestas 80-2MW	80	10	cb01-cb02-cb05
wf02	Kentis Flats	Vestas 90-3MW	30	∞	cb01-cb02-cb04-cb05
wf03	Ormonde	Senvion 5MW	30	4	cb03-cb04
wf04	DanTysk	Simens 3.6MV	80	10	cb01
wf05	Thanet	Vestas 90-3MW	100	10	cb04-cb05

Table 1: Basic information on the real-world wind farms that were used for testing.

1.2 INSTANCE TEST BED

The test that we were able to use are the combinations between the 5 instances of wind farms and the 5 sets of real cables. For simplicity reasons

and in order to eliminate mistakes, we adopted the same a numeration as the Martina Fischetti and Pisinger, 2018 article. This solution allows us to easily compare our results with the results in that research. Table ?? reports the resulting numbers of the different instances. For example we renamed the 01 wind farm as data_01.turb and the corresponding cables as data_01.cbl.

number	wind farm	cable set
01	wf01	wf01_cb01_capex
02		wf01_cb01
03		wf01_cb02_capex
04		wf01_cb02
05		wf01_cb05_capex
06		wf01_cb05
07	wf02	wf02_cb01_capex
08		wf02_cb01
09		wf02_cb02_capex
10		wf02_cb02
12		wf02_cb04_capex
13		wf02_cb04
14		wf02_cb05_capex
15		wf02_cb05
16	wf03	wf03_cb03_capex
17		wf03_cb03
18		wf03_cb04_capex
19		wf03_cb04
20	wf04	wf04_cb01_capex
21		wf04_cb01
26	wf05	wf05_cb04_capex
27		wf05_cb04
28		wf05_cb05_capex
29		wf05_cb05

Table 2: Test Bed instance numbers.

2 MATHEMATICAL MODEL

2.1 WIND FARM CABLE PROBLEM INTRODUCTION

We have studied the Wind Farms Cable Problem, which is represented by a number of wind farms in the sea that produces energy; the power production needs to be routed by some cables to the substation and then directed to the coast. To do that, each turbine must be connected through a cable to another turbine, and eventually to a substation.

The problem complexity of the cable routing problem is strongly NP-hard according to Martina Fischetti and Pisinger, 2018. They have proved that the problem is NP-hard in two formulations. First, in the case where all turbines have the same power production and the nodes are not associated with points in the plane. Second, in the case where the turbines can have different power production and are associated with point in the plane.

When designing a feasible cable routing, it's necessary to take in account a number of constraints. Here we list some of them and then we'll describe the mathematical model that realizes those constraints. Our model is based on the following requirements:

- since the energy flow is unsplittable, the energy flow leaving a turbine must be supported by a single cable.
- power losses should be avoided because it will cause revenue losses in the future.
- different cables, with different capacities and costs are available. This means that it is important to choose the right cable to minimize the costs without affecting the revenues.
- the energy flow on each connection cannot exceed the capacity of the installed cable.
- due to the substation physical layout, a given maximum number of cables, say C, can be connected to each substation.
- cable crossing should be avoided. (we will discuss this problem in the next subsection)

Let K denote the number of different types of cables that can be used and let n be the number turbines.

Definition of y_{ij} :

$$y_{ij} = \begin{cases} 1 & \text{if arc } (i,j) \text{ is constructed} \\ 0 & \text{otherwise.} \end{cases} \qquad \forall\, i,j = 1,...,n$$

$$y_{ii} = 0, \quad \forall i = 1, ...n$$

Definition of x_{ij}^k :

$$x_{ij}^k = \begin{cases} 1 & \text{if arc } (i,j) \text{ is constructed with cable type } k \\ 0 & \text{otherwise.} \end{cases} \qquad \forall\, i,j = 1,...,n \,\forall\, k = 1,...K$$

Definition of f_{ij} :

$$f_{ij} \geqslant 0, \quad \forall i,j = 1,...n$$

Objective function:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k}^{K} cost(k) \cdot dist(i,j) \cdot x_{ij}^k \tag{1}$$

Constraints:

$$\sum_{j=1}^{n} y_{hj} = \begin{cases} 1 & \text{if } P_h \geqslant 0, \quad \forall h = 1,...,n \\ 0 & \text{if } P_h = -1 \end{cases} \tag{2}$$

$$\sum_{i=1}^{n} y_{ih} \leqslant C, \quad \forall h \mid P_h = -1 \tag{3}$$

$$\sum_{j=1}^{n} f_{hj} = \sum_{i=1}^{n} f_{ih} + P_h, \quad \forall h \mid P_h \geqslant 0 \tag{4}$$

$$y_{ij} = \sum_{k=1}^{K} x_{ij}^k, \quad i,j = 1,...,n \tag{5}$$

$$\sum_{k=1}^{K} cap(k) x_{ij}^k \geqslant f_{ij}, \quad \forall i,j = 1,...,n \tag{6}$$

[magari cambiare ordine qui sotto]
The objective function 1 minimizes the total cable layout cost, where $dist(i,j)$ is the Euclidean distance between nodes i and j and $cost(k)$ is the unit cost for the cable k. Constraints 5 impose that only one type of cable can be selected for each build arc. Constraints 4 are flow conservation constraints: the energy exiting each node h is equal to the energy entering h plus the power production of the node. Constraints 6 ensure that the flow does not exceed the capacity of the installed cable. Constraints 2 impose that only one cable can exit a turbine and that no one cable can exit from the substation. Constraint 3 imposes the maximum number of cables (C) that can enter in a substation, depending on the data of the instance. The image 1 shows an example of a graphical solution to the cable routing problem.

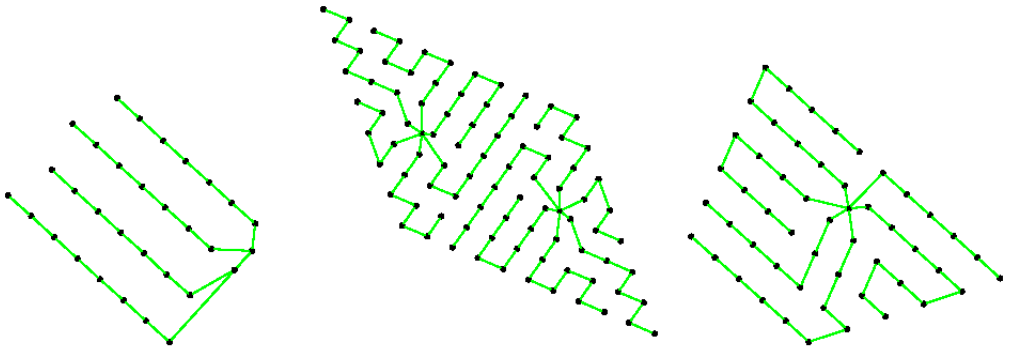


Figure 1: Example solutions to a cable routing problem

2.2 CROSSING CABLES

According to Martina Fischetti and Pisinger, 2018, an important constraint is that cable crossings should be avoided. In principle, cable crossing is not impossible, but is strongly discouraged, in practice as building one cable on top of another is more expensive and increases the risks of cable damages.

In order to evaluate if two arches cross, we used the Cramer Method: given the arc a between P_1 and P_2 and the arc b between P_3 and P_4. We define the coordinates of a general point as: $P_j = (X_i, y_i)$.
Using the Cramer method we have:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + \lambda \begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \end{pmatrix} \quad \lambda \in]0,1[$$

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_3 \\ y_3 \end{pmatrix} + \mu \begin{pmatrix} x_4 - x_3 \\ y_4 - y_3 \end{pmatrix} \quad \mu \in]0,1[$$

Then we evaluate if the determinant is equal to zero value. By using a calculator we will check if the determinant is smaller than a constant epsilon with value $\simeq 10^{-9}$. We can have two situations:

1. if $det = 0 \quad \Rightarrow$ no crossing
2. if $det \neq 0 \quad \Rightarrow (\lambda, \mu)$ if $(\lambda \in]0,1[)$ && $(\mu \in]0,1[) \Rightarrow$ crossing

[spiegareeee]!!!

$$y(a,b) + y(c,d) \leqslant 1, \quad \forall (a,b,c,d) : [P_a, P_b] \text{ cross } [P_c, P_d]$$

3 CPLEX

3.1 PLAIN EXECUTION

We simply create the linear programming model and then we pass it to CPLEX for the optimization. The performances, as we will notice for all the methods, depend on the instance. We noticed the power of CPLEX which finds the optimal solution in few seconds, and in the meantime we discovered that some instances take many hours to be solved by our machines. The main steps of our code in this phase are:

- Read the input files and the command line parameter and parse them;
- Memorize turbines and cables;
- Develop the specific linear programming model;
- Call CPX_INT_OPT that optimizes the instance;
- Ask to CPLEX the optimal function;
- Print and graph it;

3.1.1 Relaxed Mode

Sometimes there are constraints that risk to block the improvement of the solution or make CPLEX take longer before finding a feasible solution.
To avoid those situations, it is possible to *"relax"* some constraints in order to speed up the process of searching for the first solution. The main steps are to add a slack variable $\geqslant 0$ in the model and then add this variable in the objective function multiplied for a constant reasonably large. In this way, even if CPLEX could find a wrong initial solution, it is probable that this solution will rapidly improve and be corrected. Anyway, the choice to relax some constraint should not affect the optimal solution because of the very high impact on the objective function.
In our case we found three constraints that could be relaxed: the maximum number C of edges entering in a substation, the flux losses and the need to have one big connected component that connects all the arcs (for some heuristic algorithm).
The first constraint can be relaxed in this way (see the result in image 2):

$$f_{obj}^{R_1}(x, y, k) = f_{obj}(x, y, k) + \text{BIG\_M\_CABLE} \cdot s$$

$$c \geqslant \sum_{i=1}^{n} y_{ih} - s \quad \forall\, h : P_h = -1$$

The second constraint, about the flux losses, can be relaxed in this way (see the result in image 3):

$$f_{obj}^{R_2}(x, y, k) = f_{obj}(x, y, k) + \text{BIG\_M\_CABLE} \cdot \sum_{i=1}^{n} s_i$$

$$\sum_{i=1}^{n} f_{hs} = \sum_{i=1}^{n} f_{ih} + P_h - s_h \quad \forall h = 1, ...n$$

And finally it is possible to add to the relaxed constraint $f_{obj}^{R_2}(x, y, k)$ also the third option, simply adding the following constraint (see the result in image 4):

$$\sum_{j=1}^{n} y_{ij} < 1 \quad \forall i = 1, ...n$$

We listed the three different possibility that we can activate in our algorithms. The second and the third options come out of thinking about heuristic algorithm, while the first option can be applied in all the algorithms.

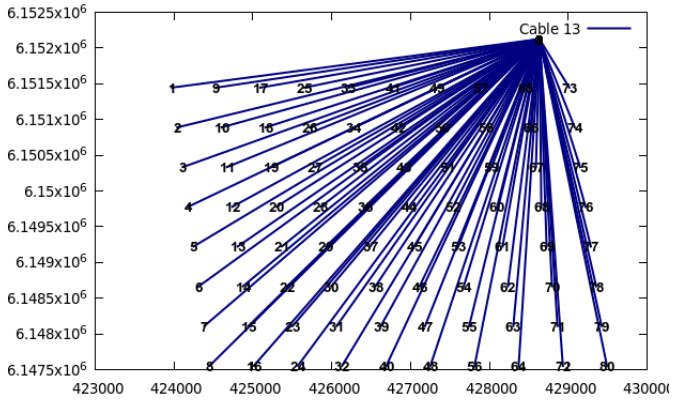


Figure 2: Example relax 1

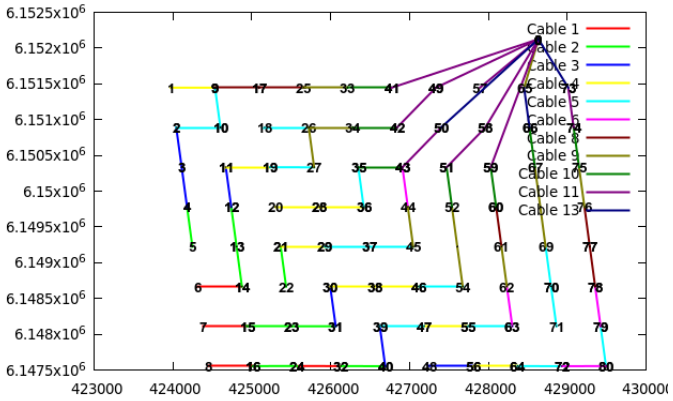


Figure 3: Example relax 2

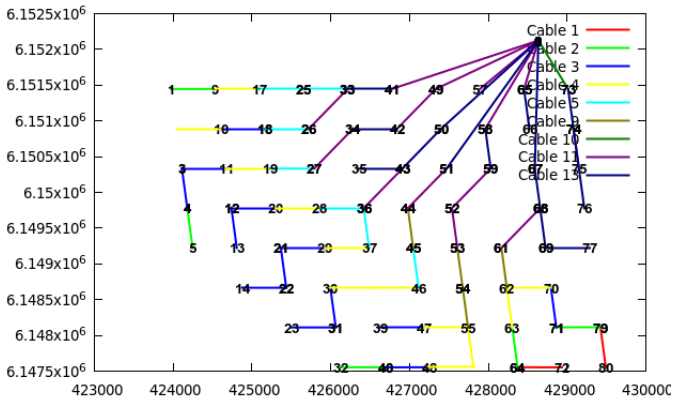


Figure 4: Example relax 3

3.1.2 CPLEX Heuristics-Params

The CPLEX code contains some heuristic procedures. Being integrated into branch & cut, they can speed the final proof of optimality or provide a suboptimal but high-quality solution in a shorter amount of time than by branching alone. With default parameter settings, CPLEX automatically invokes the heuristics when they seem likely to be beneficial. However, it is possible to adapt them to our specific case and instances by changing the frequency of activation of these procedures by setting some CPLEX parameter. We mainly analyzed two of them:

- **RINS**: it means *Relaxation induced neighborhood search* and it is a heuristic that explores a neighborhood of the current incumbent solution to try to find a new, improved incumbent. In the initial part the process does not change, but as soon as a solution is found the RINS heuristic tries to improve it with more frequency. It is possible to infer that the RINS method has been used in a CPLEX step when in the logs there is a '*' near to the number. We set the *-rins* param to 5 in all the tests.
- **POLISHING**: this heuristic tries to modify some variables of a (good) solution to improve it. It is possible to set a condition that enables this method, in order to avoid premature usage of this method, which that can lead to wasted time and performances. We decided not to change this parameter in our tests.

3.2 LAZY CONSTRAINTS METHOD

The generated constraints are often too much and risk to block the solution for a long time if we added to the model statically. Instead of adding systematically all the constraints in the model at the beginning, we generate them "on the fly" when they are violated by the Branch and Bound process. In this way CPLEX will check those constraints only when a solution is created. In the case that some constraint is violated, CPLEX will add the corresponding constraint before the incumbent update.
The CPLEX command to add a set of constraints is *CPXAddLazyConstraints*. This technique decreases the efficiency of the CPLEX pre-processing and

sometimes the computation time of the solution is really high, but generally gives good results.

3.3 LOOP METHOD

In this method the CPLEX execution will be reiterated until the optimal solution is not found. From this feature comes his name.
The model that we initially use do not include the no-crossing constraints and it is not possible to set all of them at one time. So in this method, we add time by time only the necessary constraints after each loop of the method. In particular after each loop we must verify that the chosen cables do not intersect.
This method is mathematically correct, but seems inefficient. However, nowadays the power of CPLEX pre-processing allows the Loop Method to be considered. We have also implemented a variant of this method that can stop the execution in some cases, even if the optimality has not yet been reached. This optimization comes from the fact that often CPLEX spends a lot of time demonstrating the optimality of a solution. However, maybe this solution uses some crossing cables. The stopping conditions can be several: stop at the first solution (not so good in our case because often it is the "star solution"), stop when the gap is less than a fixed percentual or stop after a timelimit.
In our solution we chose a variant of the timelimit condition. We have defined *timestart* and *timeloop* parameters: when the algorithm starts CPLEX runs for *timestart* seconds searching for a good starting solution, then the loops will last *timeloop* seconds, until the real *timelimit* expires. The main reason of this choice is to seek for a good solution before the starting of the real Loop method.
This technique solves consecutively more complex models. It is important to note that, if no new constraint is added to the model, CPLEX can restart from the last solution found saving a lot of time. One limitation of this method is that at the end it is possible (especially in the case of few loops iterations) to have crossing edges because initially the CPLEX solver is launched without constraints.

3.4 LAZY CALLBACK METHOD

CPLEX supports callbacks so that it is possible to define functions that will be called at crucial points in the application. In particular, lazy constraints are constraints that the user knows are unlikely to be violated, and in consequence, the user wants them applied lazily, not before needed. Lazy constraints are only (and always) checked when an integer feasible solution candidate has been identified, and any of these constraints that is violated will be applied to the full model.
In this method we have added the no-crossing constraint calling the *lazy constraint callback* (named "LazyConstraintCallBack") each time the solver find an integer feasible solution.

The algorithm starts with the creation of the model, and the lazy constraint callback is installed to make CPLEX call it when needed. Then, the CPLEX solver starts to resolve the model applying the branch-and-cut technique. When CPLEX finds an integer feasible solution, the lazy constraint callback is called. The solution that we have now is an integer solution of the problem where, perhaps, some of the arcs intersects. Hence, starting from this solution we have to check if there are cables crossing and, if there are any, we have to add the corresponding constraints. In this way, only necessary constraints are added and CPLEX will make sure that these constraints will be satisfied before producing any future solution of the problem.
The algorithm terminates when CPLEX finds the optimal integer solution without crossing, or when the *timelimit* is reached.
Different from the loop method, CPLEX generates the optimal solution only for the final model, adding the constraints during the resolution of the problem. This tendentially leads to the addition of more constraints in respect to the loop method, hence in some cases, leading to a slower execution.

3.5 RESULTS

With the objective to compare our algorithms we have executed various instances of the Wind Farm Cable Problem analyzing their solution after five and ten minutes for each method. All the instances have 30, 80 or 100 turbines with the number of cables that varying by 3 to 13 types. In this table we can see the result of this execution. As we can see the first method, the basic method with CPLEX that contains also all the cross constraints does not work bad but it initially has a lot of problem to compute the model because sometimes we have too many constraints, in some case our computers are blocked (in the table: nan). To escape from this problem we set the no cross constraints as lazy constraint using the pool of lazy constraints of CPLEX, the lazy callback and the loop method. As we can see there are not more differences between the first two method, nothing that can be attributed to one method considering the performance variability. Also if we consider the mean of gaps between the solution with the CPLEX pool and the lazy callbacks, the method with CPLEX pool has the best results in ten minutes while in five minutes win the method with lazy callbacks. The method to compute the gap that we used is:

$$gap = \frac{(solution1 - solution2)}{(solution1 + solution2)}$$

with solution 1 and 2 relative to the same instance.The sign of the gap tell us which the method wins.
While the loop method seems to be the worst method and have a problem, the loop method must be start from a good solution before than it iterate and removes the cross, so we have, as first, compute a solution with half of time of the total execution of method.

Table 3: CPLEX based methods results with *timelimit* 5 minutes

Instance	**CPX basic**		**CPX lazy const**		**CPX loop (60s)**		**CPX lazy callback**	
	time	solution	time	solution	time	solution	time	solution
data_01	300	2.45E+07	300	2.05E+07	300	2.13E+07	300	1.98E+07
data_02	nan	nan	300	2.16E+07	300	2.45E+07	300	4.02E+09
data_03	300	2.58E+07	300	7.02E+09	300	2.35E+07	300	2.35E+07
data_04	301	3.14E+07	300	2.48E+07	300	2.66E+07	300	2.48E+07
data_05	300	1.00E+08	300	7.02E+09	300	7.00E+10	300	2.40E+07
data_06	nan	nan	300	4.02E+09	300	3.49E+07	300	3.02E+09
data_07	3	8.56E+06	4	8.56E+06	4	8.56E+06	300	8.66E+06
data_08	8	8.81E+06	9	8.81E+06	7	8.81E+06	192	8.81E+06
data_09	9	1.01E+07	31	1.01E+07	2	1.01E+07	3	1.01E+07
data_10	6	1.03E+07	8	1.03E+07	10	1.03E+07	24	1.03E+07
data_12	13	8.60E+06	2	8.60E+06	3	8.60E+06	300	8.58E+06
data_13	7	8.93E+06	5	8.93E+06	4	8.93E+06	9	8.93E+06
data_14	16	1.02E+07	26	1.02E+07	3	1.02E+07	12	1.02E+07
data_15	14	1.03E+07	7	1.03E+07	5	1.03E+07	34	1.03E+07
data_16	225	8.05E+06	20	8.05E+06	21	8.05E+06	30	8.05E+06
data_17	88	8.56E+06	142	8.56E+06	300	8.56E+06	300	8.56E+06
data_18	300	8.36E+06	140	8.36E+06	300	8.36E+06	300	8.36E+06
data_19	102	9.18E+06	161	9.18E+06	300	9.27E+06	46	9.21E+06
data_20	300	1.99E+08	300	1.20E+10	300	4.50E+10	300	1.30E+10
data_21	300	1.17E+08	300	8.04E+09	300	3.10E+10	117	1.50E+10
data_26	300	2.34E+07	300	7.03E+09	300	9.00E+10	300	2.29E+07
data_27	nan	nan	300	2.50E+07	300	2.65E+07	299	2.39E+07
data_28	nan	nan	300	1.70E+10	300	3.35E+07	204	1.20E+10
data_29	nan	nan	300	4.65E+07	300	4.65E+07	2	9.30E+10

Table 4: CPLEX based methods results with *timelimit* 10 minutes

Instance	**CPX lazy constraint**		**CPX loop (120s)**		**CPX lazy callback**	
	time	solution	time	solution	time	solution
data_01	600	1.95E+07	600	2.07E+07	600	1.98E+07
data_02	600	2.38E+07	600	2.15E+07	600	2.49E+07
data_03	600	2.27E+07	480	2.27E+07	577	2.31E+07
data_04	600	2.45E+07	600	2.54E+07	599	2.48E+07
data_05	600	2.46E+07	600	2.46E+07	538	2.39E+07
data_06	600	2.74E+07	480	2.74E+07	587	3.00E+11
data_07	3	8.56E+06	600	8.56E+06	16	8.56E+06
data_08	19	8.81E+06	3	8.81E+06	44	8.81E+06
data_09	22	1.01E+07	3	1.01E+07	5	1.01E+07
data_10	12	1.03E+07	5	1.03E+07	85	1.03E+07
data_12	2	8.60E+06	3	8.60E+06	1	8.60E+06
data_13	6	8.93E+06	4	8.93E+06	7	8.93E+06
data_14	4	1.02E+07	3	1.02E+07	27	1.02E+07
data_15	7	1.03E+07	8	1.03E+07	9	1.03E+07
data_16	14	8.05E+06	22	8.05E+06	82	8.05E+06
data_17	149	8.56E+06	63	8.56E+06	67	8.56E+06
data_18	213	8.36E+06	71	8.36E+06	491	8.36E+06
data_19	211	9.18E+06	26	9.18E+06	342	9.18E+06
data_20	600	4.01E+07	360	4.01E+07	110	1.70E+12
data_21	600	4.70E+07	360	4.70E+07	96	1.30E+12
data_26	600	2.27E+07	480	2.27E+07	399	2.26E+07
data_27	600	2.42E+07	600	2.37E+07	600	2.45E+07
data_28	600	2.84E+07	240	2.84E+07	600	5.00E+11
data_29	600	3.09E+07	600	3.09E+07	600	3.00E+12

4 MATHEURISTIC METHODS

Given that the CPLEX solver is really optimized, we apply the heuristic in the model writing. In this way the model that we'll give to CPLEX should be theoretically easier to solve.

4.1 HARD FIXING

The first matheuristic algorithm that we have implemented relied on the hard variable fixing approach. The main idea is to use a *black-box* solver which receives the input data and quickly generates a first solution. Once the initial solution has been found, some of its variables are fixed and then the method is iteratively reapplied on the restricted problem resulting from fixing: the *black-box* solver is called again, a new target solution is found, some of its variables are fixed, and so on. The choice of which variables have to be fixed is arbitrary, so the edges are chosen with uniform probability.
Each time, before applying the CPLEX solver, the algorithm fixes some variables of the last solution obtained; an important parameter that influences the performances of the CPLEX solver is the number of edges that have been fixed in each loop. If the number of fixed edges is high, CPLEX will find a solution more quickly. On the other hand, if the number of fixed edges decreases, CPLEX is more free to find new improvements.
We chose to fix all the arcs with probability 0.5 for each one. Using the hard fixing technique, and in general fixing some variables, CPLEX becomes faster because the number of arcs decreases. The number of arcs decreases in three ways:

1. some of them are fixed
2. because we "delete" all the arcs exiting from a node which already has an exiting arc
3. because we "delete" all the arcs crossing with those already fixed

We discovered that in the first CPLEX solutions very often appears the "star" solution, see image 2. The "star" is a shape of routing cables in which all the turbines are directly connected to the substation; in almost all the instances it is not a good solution because of the long cables. This fact can be a problem for the Hard Fixing technique because it is most likely to choose fixed cables far away from the optimal solution. To avoid this situation we have defined a *timestart*: when the algorithm starts CPLEX runs for *timestart* seconds searching a good starting solution, then it starts the real Hard Fixing method until the *timelimit* expires.

4.1.1 RINS Hard Fixing

This solution is like the classic Hard Fixing solution with the added condition that when we have to choose a fixed edge, that edge must be present in the actual best solution. Then all the probabilistic mechanisms do not change. This is something like "doing hard fixing over the RINS condition".

4.2 SOFT FIXING

This method, also called *local branching*, given a solution called y^{REF}, fixes at least a percentage of the arcs of that solution, and repeats the execution searching the best choice for the others. A critical issue of variable fixing methods is related to the choice of the variables to be fixed at each step and wrong choices are typically difficult to detect. In this sense, the purpose of the soft fixing is to fix a relevant number of variables without losing the possibility of finding feasible solutions.
A possible implementation is, given the solution y^{REF} represented by an array of zeros and ones, given a generic solution y and given a constant K:

$$y^{REF} = (0, 1, 0, 1, 1, ...)$$

$$\sum_{(i,j):y_{ij}^{REF}=0} y_{ij} + \sum_{(i,j):y_{ij}^{REF}=1} (1 - y_{ij}) \quad \leqslant K$$

It represents the Hamming distance between y and y^{REF}; in practice the constraints allows one to replace at most K edges of y^{REF}.
Then, in our implementation, the algorithm starts producing a heuristic solution Y, adds the local branching constraint to the MIP model and solves it using CPLEX.
In our solution we decided to start with $K = 3$; each time in increments of two units until it reaches the maximum of 20: then it restarts from 3. It is important that the number K changes during the different execution to allow the *local branching* to explore different solution types. We decided those specific numbers after simple test on some instances.
Soft fixing avoids a rigid fixing of the variables in favor of a more flexible condition. This allows the new solution to "move" from the older one fixing at each iteration some random arcs and moving the other looking quickly for a better solution.
This is the symmetric version of this method because it considers equally the 0-1 and the 1-0 flips.

4.2.1 Asymmetric Soft Fixing

In this case we consider only the flips from 1 to 0:

$$\sum_{(i,j):y_{ij}^{REF}=1} (1 - y_{ij}) \quad \leqslant K$$

$$\sum_{(i,j):y_{ij}^{REF}=1} y_{ij} \quad \geqslant \sum_{(i,j):y_{ij}^{REF}=1} 1 - K$$

$$\sum_{(i,j):y_{ij}^{REF}=1} y_{ij} \quad \geqslant n-1-K$$

This method is more convenient from the graphical point of view.

4.2.2 Soft Fixing RINS (asymmetric)

This method is a variant of the Soft Fixing. RINS algorithm is an heuristic that explores a neighborhood of the current incumbent solution to try to find a new and improved incumbent. In practice it compares to the variable values of the good solutions and when the (??? spiegare come funziona). In the asymmetric case it looks only for the 1 values.
It is possible to realize also the Symmetric RINS that consider also the 0 values. Anyway, we have discarded this option from the tests because for this specific practice case the selected arcs are more important than the ones not selected.

4.3 RESULTS

The MathEuristic method was born to have a better solution with less time, unless certificate the optimality of the solution. The purpose is use the CPLEX normal execution inserting and removing some conditions. In the table we can see a Hard and Soft Fixing method with and without using the RINS strategy.
The best result that we have obtained is with the Hard Fixing with the RINS strategy but also here the result obtained is similar. While if we compare this results with the precedent we see that the MathEuristic method return the best results.

Table 5: Matheuristic methods results with *timelimit* 10 minutes

Instance	**Hard Fixing**		**Hard Fixing RINS**		**Soft Fixing Asymmetric**		**Soft Fixing RINS**	
	time	solution	time	solution	time	solution	time	solution
data_01	349	1.89E+07	359	1.97E+07	355	1.89E+07	478	1.95E+07
data_02	215	2.02E+09	348	2.16E+07	541	2.15E+07	364	2.15E+07
data_03	304	2.30E+07	275	2.28E+07	178	2.28E+07	319	2.43E+07
data_04	419	2.45E+07	600	2.49E+07	299	2.53E+07	279	2.53E+07
data_05	83	9.02E+09	360	2.41E+07	571	2.42E+07	357	2.50E+07
data_06	414	2.71E+07	600	2.76E+07	540	2.49E+07	484	2.52E+07
data_07	8	8.30E+06	234	8.56E+06	8	8.42E+06	9	8.23E+06
data_08	57	8.81E+06	236	8.81E+06	471	8.81E+06	508	8.81E+06
data_09	4	1.01E+07	25	9.88E+06	6	1.01E+07	5	1.01E+07
data_10	528	1.03E+07	35	1.03E+07	11	1.03E+07	264	1.03E+07
data_12	2	8.60E+06	600	1.09E+07	239	8.60E+06	56	8.60E+06
data_13	19	8.93E+06	21	8.93E+06	20	7.40E+06	23	8.13E+06
data_14	40	9.73E+06	243	1.02E+07	24	1.01E+07	186	1.02E+07
data_15	61	1.03E+07	95	1.03E+07	101	1.03E+07	232	1.03E+07
data_16	116	8.05E+06	65	8.05E+06	37	8.05E+06	363	8.05E+06
data_17	291	7.93E+06	540	8.56E+06	62	8.56E+06	90	8.56E+06
data_18	60	8.36E+06	120	8.36E+06	403	8.36E+06	404	8.36E+06
data_19	305	9.21E+06	600	9.49E+06	346	8.35E+06	358	1.30E+10
data_20	300	1.10E+10	300	1.60E+10	296	1.50E+10	275	3.04E+09
data_21	280	5.04E+09	460	2.04E+09	341	4.04E+09	600	3.04E+09
data_26	558	2.28E+07	600	2.24E+07	479	2.24E+07	600	2.41E+07
data_27	479	2.26E+07	360	2.38E+07	360	1.54E+07	600	1.54E+07
data_28	300	4.03E+09	281	2.03E+09	419	2.70E+07	356	3.03E+09
data_29	538	2.03E+09	360	9.20E+10	592	7.03E+09	360	9.10E+10

5 HEURISTIC METHODS

As we have said, the WFCP belongs to the class of NP-Hard problems. Therefore, when the number of nodes is too high, we cannot obtain an optimal solution in a feasible time. For this reason we have decided to implement some algorithms that, instead of solving the mathematical model, use heuristic methods to find a solution of the problem. These methods, relying on the characteristic structure of the problem, can compute quickly a good solution that is not guaranteed to be optimal. Then, in the following section, we are going to describe some heuristic methods that iteratively execute a specific procedure trying to obtain a consecutively better solution.

5.1 PRIM–DIJKSTRA

The Prim-Dijkstra algorithm was developed in 1930 by Czech mathematician Vojtěch Jarník and later rediscovered and republished by computer scientists Robert C. Prim in 1957 and Edsger W. Dijkstra in 1959.
In computer science, Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means that it finds a subset of the edges that forms a tree which includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex Cheriton and Tarjan, 1976.
In our solution we personalized a little bit the algorithm. In order to avoid that only one edge is connected with the substation, we have decided a fake distance between turbines and the substation so that there are more connection between the turbines and the substation.
It is important to note that the resulting graph could not respect the C constraint and can have crossing cables.

5.2 GRASP

The GRASP algorithm, that means *Greedy Randomized Adaptive Search Procedures*, is a metaheuristic algorithm first introduced by Feo and Resende (1989). GRASP typically consists of iterations made up from successive constructions of a greedy randomized solution and subsequent iterative improvements of it through a local search. GRASP is a multi-start metaheuristic.
It consists of two phases: greedy randomized adaptive phase (generating some solutions) and local search (finding a local minimum).
The first phase is further composed by two parts, a greedy algorithm and a

probabilistic selection.
The *greedy algorithm* always makes the choice that looks the best at the moment, that in our case is to find the best N = 10 possible edge choices, so the N closer turbines. The *probabilistic part* consists of flipping a coin and if it comes out heads choose the best of the 10 edge choices. Otherwise, randomly pick another solution. This probabilistic part has the objective to diversify the solutions of the greedy algorithm, adding randomness to an algorithm. This can lead to some good solutions that a deterministic procedure could not obtain. Note that if all the coins come out heads, then this algorithm would take exactly the same decisions of the Dijkstra algorithm.
Once we have found a solution using the first phase, we have to reach the local optimum. For this process we use the *1-Opt* technique, described in the next subsection (5.2.1), which finds the local optimum from an existing solution.
Our algorithm executes iteratively all of these steps, and at the end of the time this algorithm will return the best solution found.
This intuitive algorithm can mitigate the threat to finish always in a local minimum (*1-Opt*) with the randomness added by the greedy randomized phase.
With this algorithm, there isn't the mathematical proof that the optimum will be reached, but the probability of *"to not reach the optimum"* is infinitely close to 1 (obviously considering an infinite number of executions).

5.2.1 1-Opt

It belongs to the category *Refining Heuristic Algorithm*, therefore it tries to improve an already existing solution.
Basically it tries to substitute an arc with another one that reduces the cost of the objective function.
Obviously the end of this algorithm will be inside a local minimum; therefore the results of this procedure will be a local optimum.
The images 5, 6 represent one step of this algorithm.

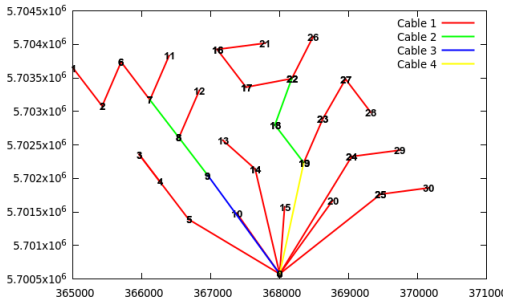


Figure 5: Example 1-opt operation step 1

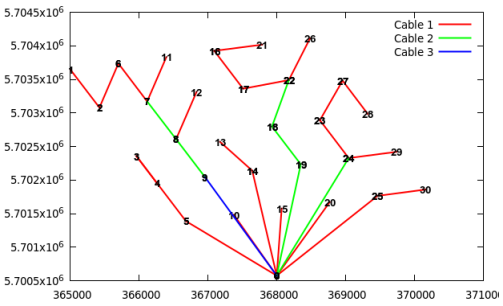


Figure 6: Example 1-opt operation step 2

5.3 TABU SEARCH

Tabu search, proposed by Fred Glover in 1986 and formalized in 1989, is a metaheuristic search method that guides a local search procedure to explore the solution space beyond local optimality. Tabu search is based on the premise that problem solving, in order to qualify as intelligent, must incorporate adaptive memory and responsive exploration. Since local choices are guided by information collected during the search, Tabu search contrasts with memoryless procedures that heavily rely on semi-random processes that implement a form of sampling. The emphasis on responsive exploration in Tabu search, whether in a deterministic or probabilistic implementation, derives from the supposition that a bad strategic choice can often yield more information than a good random choice, and therefore provides a basis for progressively improved strategies that take advantage of search history.
In our algorithm Tabu search means to make a move and consider the opposite move *"Tabu"*. In this way the algorithm tries to avoid the local optimum In our solution the algorithm start from (???) and applies a *1-OPT* local search until a local optimum is reached. Then, to try to "escape" from the local optimum, the algorithm makes a sequence of moves that deteriorate the current objective function. (quali mosse ???). At each move, whether it is improving or worsening the previous solution, the algorithm stores the edges that are involved in the exchange in a data structure, marking them as *tabu*, which means that they can not be involved again in the future moves. This type of memorization, called *recency-based memory*, keeps track of solution attributes that have changed during the recent past.
The capacity of the algorithm to go away from the local optimum is determined by the number of moves that the memory structure can store, and it directly influences the diversification stage. This parameter called *tabu tenure* is really difficult to decide a-priori its size for two reasons: if the *tabu tenure* is too high then the risk is to "freeze" the algorithm, because only a moves are allowed, and, if it is too low then it will is not assure that the algorithm can escape from all the local optimums. So, the choice of *tabu tenure* needs to be tested in some runs of the algorithm depending on the instance that has to be solved.(cosa abbiamo scelto noi??)
It can happen that some (c'è da spiegare qualcosa per quanto riguarda il fatto che quando ci sono le mosse peggiorative la nostra soluzione ritorna infeasible a causa dei crossing e delle perdite di flusso???)

5.4 ANT COLONY ALGORITHM

Ant Colony Optimization (ACO) is a heuristic optimization algorithm. The main idea is to follow what the ants do when they find food. An ant moves randomly in the world leaving pheromones and if they do not continue to find anything on the path then the pheromones decay over time. When they finds food, they take it and return to their anthill. They then retrace the path that they previously took, doubling the normal quantity of pheromones. The more pheromones on the path the higher the chance that a new ant follows this path. When another ant follows this path, it reinforces

the line of pheromones.
With this same logic we can create algorithm 1 for our problem. Initially every edge of the solution is chosen with the same probability, starting from this solution the algorithm simulates the leaving of pheromones, incrementing the probability of the edges chosen in base on the quality of solution and decreasing the other edges of a fixed percentage to simulate the decay over the time of pheromones. Then the algorithm repeats the choice of the solution basing of the new probability. Doing so, after some repetition of the loop the probability to choose an edge of the optimal solution should be greater than the other edges.

Algorithm 1 : Ant Colony Optimization

Input: Graph $G = (E, V)$, function cost: $P(E) \to \mathbb{R}$
Data: Pheromone Values $P_E : E \to \mathbb{R}$, Current solution $C \subseteq$
Output: $S \subseteq E$
Initialize pheromone values
repeat
  $C \leftarrow$ findPath(G, P_E)
  $P_E \leftarrow$ updatePheromones(C, P_E)
  if cost(C) < cost(S) **then**
    $S \leftarrow C$
until *time limit reached*
return S

There are a lot of way to implements the logic of the algorithm as we have found in Nedlin, 2017, every method for our problem is based on how is implemented the findPath function. We have chosen to implement the ant colony algorithm based on the Kruscal's algorithm.
The Kruscal algorithm Kruskal, 1956 is meant to find the minimum coating tree, that links all nodes and that has the sum of the weight of its edges as the minimum. The main idea is to order the edges with the increasing weights and choose the new edge to add to the tree that one that has the minimum cost and does not make cycle with the already selected edges.
We realized the Ant algorithm in the Kruscal approach: as a basis we used again the common ACO algorithm before presented but with a modified FindPath function. In this approach we selected the new edge to add to the tree. So first we choose randomly a node, and then we select, using the probability given by the pheromones, an outer edge from the node chosen. The outer edge must satisfy some conditions:

1. Do not crossing, this condition can be forced if there aren't others choice
2. Do not cycles

This operation repeats until all nodes are linked.

Algorithm 2 : FindPath in Kruskal Approach

Input: Graph $G = (E, V)$, Pheromone Values P_E
Data: Current Node c, Choice List L
Output: $T \subseteq E$
repeat
  L.CLEAR()
  // select all valid neighbors
  forall $(c, v) \in E \setminus \{e \in E \mid T \bigcup \{e\}$ *contains a circle* $\}$ **do**
    if *c or v are not connected to a substation in* (V, T) **then**
      L.INSERT*(c, v)*
  if L *is not empty* **then**
    Select a random edge e = { c, v } from L using P_E as weight
    $T \leftarrow T \bigcup \{e\}$
until L *is empty*

5.5 RESULTS

The Heuristic method are method that does not use CPLEX, with simple strategy try to improve their solutions. In this table we can see the result obtained with the tabu search, the ant algorithm and the multistart. The ant algorithm, except in few cases, is that which have the worst results.
As we can see from the results this method does not work very well, this is because with a not enough iteration an edges that is in the optimal solution could have a low level of pheromones caused by the fact that the edge could be chosen in bad solutions derived by the choice of other edges and therefore it remain penalized.
While between the Tabu search method and the multi start method, the Tabu search has the best result, also because the loop method is almost rudimentary, we have implemented it as a Dijkstra algorithm with the variation introduced by GRASP strategy and use 1-Opt move to decrease the objective function. If we compare the Tabu search with the Hard Fixing with the RINS strategy we can see that the Hard Fixing has the best result, but the results are not so different therefore this shows that the Tabu search is still a good method.

Table 6: Heuristic methods results with *timelimit* 10 minutes

Instance	**Tabu Search**		**GRASP**		**Ant Algorithm**	
	time	solution	time	solution	time	solution
data_01	600	2.22E+07	600	1.40E+12	600	2.24E+08
data_02	600	2.42E+07	600	1.30E+12	600	2.62E+07
data_03	600	2.65E+07	600	1.02E+14	600	2.85E+07
data_04	600	2.96E+07	600	1.35E+14	600	3.20E+07
data_05	600	2.73E+07	600	9.02E+11	600	2.30E+08
data_06	600	2.71E+07	600	1.20E+12	600	2.97E+07
data_07	600	9.53E+06	600	4.18E+08	600	9.78E+06
data_08	600	9.31E+06	600	1.01E+11	600	9.50E+06
data_09	600	1.10E+07	600	6.04E+11	600	2.13E+08
data_10	600	1.19E+07	600	3.01E+11	600	1.22E+07
data_12	600	8.93E+06	600	1.01E+11	600	1.00E+07
data_13	600	9.65E+06	600	1.02E+09	600	9.51E+06
data_14	600	1.07E+07	600	1.94E+07	600	1.08E+07
data_15	600	1.08E+07	600	3.00E+11	600	1.22E+07
data_16	600	8.55E+06	600	3.02E+11	600	9.07E+06
data_17	600	9.15E+06	600	1.01E+11	600	9.59E+06
data_18	600	9.27E+06	600	1.02E+09	600	9.68E+06
data_19	600	9.99E+06	600	2.01E+11	600	1.11E+07
data_20	600	2.03E+11	600	3.79E+13	600	2.02E+11
data_21	600	2.03E+11	600	6.30E+13	600	2.01E+11
data_26	600	2.43E+07	600	1.40E+12	600	2.48E+07
data_27	600	2.55E+07	600	1.30E+12	600	2.57E+07
data_28	600	2.01E+11	600	1.77E+14	600	2.01E+11
data_29	600	2.01E+11	600	2.20E+14	600	2.01E+11

Appendices

A CPLEX

ILOG CPLEX is a tool for solving linear optimization problems, commonly referred to as Linear Programming (LP) problems, of the form:

$$\begin{aligned} \textit{Maximize (or minimize)}: \quad & c_1x_1 + c_2x_2 + c_nx_n \\ \textit{Subject to}: \quad & a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n \sim b_1 \\ & a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n \sim b_2 \\ & ... \\ & a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n \sim b_m \\ \textit{With these bounds}: \quad & l_1 \leqslant x_1 \leqslant u_1 \\ & ... \\ & l_n \leqslant x_n \leqslant u_n \end{aligned}$$

Where $\sim$ can be $\leqslant$, $\geqslant$ or $=$, and the upper bounds u_i and lower bounds l_i may be positive infinity, negative infinity, or any real number.
In particular, we have used the CPLEX Callable Library, which is a C library, that allows the programmer to embed CPLEX optimizers in application written in C. Thus, we have used CPLEX as a tool for solving Mixed Integer Programming problems since our Wind Farm Cable Problem instances.
In this appendix we want describe the main method that we use to create the CPLEX environment, to instantiate a LP problem and populate a problem object.

A.1 INITIALIZATION OF CPLEX ENVIRONMENT

The first thing to do in an application that uses CPLEX is create a CPLEX environment:

- *CPXENVptr env = CPXopenCPLEX(&error)*: initializes a CPLEX environment and returns a pointer to it; the parameter is a pointer to an integer where an error code is placed by the routine.

Then a CPLEX problem object must be created:

- *CPXLPptr lp = CPXcreateprob(env, &error, "Name of problem")*: this method returns a pointer to the CPLEX problem object. The problem that is created is an LP minimization problem with zero constraints, zero variables and an empty constraint matrix.

A.2 POPULATION OF A PROBLEM

Now we have initialize the environment and we must set the objective function and the constraints of our model. To add a variable in the our objective function we must use:

- *CPXnewcols(env, lp, ccnt, &obj, &zero, &ub, &type, cname)*: adds ccnt empty columns to the specified CPLEX problem object lp. For each column, we must specify the objective coefficient, the lower and upper bounds, the variable type and the name of the variable.

Then we have to add the constraints:

- *CPXnewrows(env, lp, rcnt, &rhs, &sense, NULL, cname)*: adds rcnt empty constraints to a specified CPLEX problem object lp; for each row, we specify the right hand side value, sense, range value (NULL), and name of the constraint; the constraint coefficients in the new rows are zero.

To set the coefficients of the constraints in a row we use:

- *CPXchgcoef (env, lp, row, column, coeff)*: changes a single coefficient in the constraint matrix in the position specified by (row, column).

A.3 CPLEX EXECUTION AND SETTING CPLEX PARAMETERS

Now we must to start the execution of CPLEX and get the result.
Before the CPLEX starts we can set some parameters relative to its execution:

- *CPXsetTYPEparam(env, CPX_PARAM_XXX, value)*: this function depends by the type of parameter, its type must substitute TYPE in the invoketion; the parameter that will be setted is the second parameters of the function and the last is the value at it will be setted.

Now we can execute the CPLEX to solve our problem:

- *CPXmipopt(env,lp)*: finds a solution to the problem lp.

Then get the informations about the solution with:

- *CPXgetobjval(env, lp, &sol_value)*: writes in sol_value the solution objective value.
- *CPXgetx(env, lp, sol , 0, ncols-1)*: writes in sol the solution variables values.

Before the end of execution we must free the space occupied by the problem with:

- *CPXfreeprob(env, &lp)*: that free the space occupied by the lp.
- *CPXcloseCPLEX(&env)*: that release the pointer relative to the CPLEX environment.

B SHELL SCRIPT

The collection of the results is an important part of our software development. Gives us the possibility to run the same code on different instances, to collect the results in a accessible way, to compare results obtained with different CPLEX parameters and finally gives us the convenience to store all the result of a run in the same folder. For those reasons when we were still at the beginning of the development we create a shell script that automates the process. It is composed by some parts:

1. Make: compiles the code at the actual state.
2. Creates the plot folder and the results folder named with the actual timestamp.
3. Defines the the command line parameter in the *settings* variable.
4. Starts a cycle iterating over all the .turb files in the data folder, that represent all the instances. For each iteration it executes the wfcp script and saves the logs in the right folder giving them a name that associate it to the instance. The *cSub* variable is set in order to use the correct C for each instance.
5. Order some files, also the .png results if present.
6. Creates the settings.txt file with the execution settings of that run.
7. Creates the results.csv file that collects all the instances results coming from the logs.

The script detects the working directory from which the script is launched so it is adaptable to different environment if the folder structure is the following:

- main directory
- data: contains all the .turb and .cbl file
- runs: contains the results
- src: contains the script files and the multi_wfcp.sh file.

Here the code of the *multi_wfcp.sh* file:

Listing 1: Shell multi_wfcp.sh script

```
# WFCP
make
rm -r plot
mkdir plot
cd ..
dir=$(date +"%Y-%m-%d_%H-%M-%S")
mkdir runs/${dir}
path=$(pwd)
settings="--model 0 --CC 3 --time_limit 600 --relax 3 --names 1"
cd ${path}'/data'
count=0
for c in `find . -type f -name '*.turb' | cut -c 3-9 |sort`
do
      echo "---"$c
      if [ $count -le 5 ]; then
            cSub="-C 10"
      fi
      if [ $count -gt 5 ] && [ $count -le 13 ]; then
            cSub="-C 100"
      fi
      if [ $count -gt 13 ] && [ $count -le 17 ]; then
            cSub="-C 4"
      fi
      if [ $count -gt 17 ]; then
            cSub="-C 10"
      fi
      cd ${path}'/src'
      ./wfcp -ft ${path}'/data/'${c}'.turb' -fc ${path}'/data/'${c}'.cbl
          ' ${cSub} ${settings} > ../runs/${dir}/run_${c}.log
done
mv *.png '../runs/'${dir}
mv plot '../runs/'${dir}
mkdir plot
cd ${path}'/runs/'${dir}
echo ${settings} > settings.txt
grep "STAT" *.log |sort > results.csv
```

C INPUT STRING PARAMETERES

In our code we tried to parameterize all the compilation options, in order to avoid to change the code for little changes or different tests.
Here we describe all the parameter in a ready-to-use list of options:

- **fc** : input cables file
- **ft** : input turbines file
- **C** : capacity of root
- **time_loop** : time for loop in loop/heuristic method
- **time_limit** : total time limit
- **time_start** : time start to heuristic method
- **model** : model type
  0. CPLEX model
  1. Matrix model
- **rins** : rins
- **relax** : relax
  1. relax on station capacity
  2. relax on flux
  3. relax on flux + out edges

  else : no relax
- **polishing_time** : polishing time
- **gap** : gap to terminate
- **seed** : random seed
- **threads** : n threads
- **CC** : Computational Context
  0. Normal execution with no cross cable as normal constraints
  1. Lazy constraints to the model
  2. loop Method
  3. Normal execution + lazy callback
  4. Hard Fixing
  5. Soft Fixing
  6. Heuristic
  7. Heuristic Loop to have multiple solution
  8. Heuristic with 1-opt
  9. Tabu Search
  10. Multi-start

  else: Normal Execution
- **soft_fix** : Type of soft fixing
  1. Asimmetric Local Branching
  2. Simmetric Local Branching
  3. Asimmetric RINS

4. Simmetric RINS

- **hard_fix** : Type of hard fixing
  1. Random hard fixing
  2. RINS
- **times** : times to do heuristic
- **names** : 1 for more clear file names

D GNU PLOT

Gnuplot is the command-line program, which can generate two and three dimensional plots of functions, data and data fits, that we have used to plot the solutions of the Wind Cable Farm problem instances. To plot a solution obtained it, we create the script that will be used to gnuplot to plot the data. The script is create in base of the cables used, we obtain a simple script as the following:

Listing 2: Gnuplot script

```
set autoscale
set term wxt title 'cost of solution'
plot \
      'plot/plot_datastd_#cable1.dat' using 1:2 with lines lc rgb "#
          color1" lw 2 title "Cable 1",\
      'plot/plot_datastd_#cable1.dat' using 1:2:(0.6) with circles fill
          solid lc rgb "black" notitle,\
      'plot/plot_datastd_#cable1.dat' using 1:2:3 with labels tc rgb "
          black" offset (0,0) font 'Arial Bold' notitle
replot \
      'plot/plot_datastd_#cable2.dat' using 1:2 with lines lc rgb "#
          color2" lw 2 title "Cable 2",\
      'plot/plot_datastd_#cable2.dat' using 1:2:(0.6) with circles fill
          solid lc rgb "black" notitle,\
      'plot/plot_datastd_#cable2.dat' using 1:2:3 with labels tc rgb "
          black" offset (0,0) font 'Arial Bold' notitle
.
.
.
replot \
      'plot/plot_datastd_#cable3.dat' using 1:2 with lines lc rgb "#
          color3" lw 2 title "Cable 3",\
      'plot/plot_datastd_#cable3.dat' using 1:2:(0.6) with circles fill
          solid lc rgb "black" notitle,\
      'plot/plot_datastd_#cable3.dat' using 1:2:3 with labels tc rgb "
          black" offset (0,0) font 'Arial Bold' notitle
```

With *#cableX* that depends by the solution and the *#colorX* is obtained from an array that contains twenty colors, this is made to have different color for different cables.
The data file #cableX.dat contains pair of nodes that will be linked with the cable X, also this file is created at run time. This files, for any cables, are like:

Listing 3: Data file containing cables

```
node1.x node1.y
node2.x node2.y

node3.x node3.y
node4.x node4.y
.
.
.
node(n-1).x node(n-1).y
node(n).x         node(n).y
EOF
```

Once that the files are created (1 for the script and K for the cables, with K equals to the number of cables used in the solution), we manage the execution of the script by the pipe with three command:

Listing 4: Script execution

```
FILE *gp = popen("gnuplot -p", "w"); // that open gnu plot on the pipe
fprintf(gp, "load 'plot/script_plot.p'\n"); // that run the script and show
    the plot
fclose(gp); // that close the plot
```

E | PERFORMANCE PROFILE

???

F PERFORMANCE VARIABILITY AND RANDOM SEED

The performance of MIP solvers is subject to some variability that appears when changing from different computing platforms.The literature case Danna, 2008, shows us that it could happen that the same instance, with the same code is solved at the root node in one platform, and requires 1426 nodes in another. It is even possible to have different results even if we're comparing two runs in different partition of the same machine. According to Lodi and Tramontani, 2013 also the permutation of rows and/or columns of a model could lead to different performance. The performance variability is common for all the different MIP solvers. The reason of that, in a simplified view, is that the small differences of precision between different environment can lead to different branching variables. Infact the Branch and Bound approach is defined "chaotic" because little variations lead to big changes.
In literature there are some approaches to this problem and also exploiting this situation for better overall solver performance: developing pseudo-cost formulas that selects the branching variables in a more robust way or to try some different basis and evaluate which is the best alternative (this solution adds a computational overhead).
Also CPLEX offers a possible solution giving the random seed parameter that can be used to change the initialization of the random number generator that is used in some internal operations, with the purpose also to speed up the computation. The random seed enforces the random variability giving the possibility to collect statistics of N different runs for more comparable results.
For the purpose of our research the performance variability is not a central question, however it is important for two reasons. First, to understand that the same code applied to the same instance can lead to different solution paths depending on the environments. Second, because a variability means that it is possible to take different choices, that means there's a way to choose better and improve again the performances.

BIBLIOGRAPHY

Cheriton, David and Robert Endre Tarjan

1976 "Finding minimum spanning trees", *SIAM Journal on Computing*, 5, 4, pp. 724-742.

Danna, Emilie

2008 "Performance variability in mixed integer programming", in *Workshop on Mixed Integer Programming, Columbia University, New York*, 08, vol. 20.

Fischetti, Martina and David Pisinger

2018 "Optimizing wind farm cable routing considering power losses", *European Journal of Operational Research*, 270, 3, pp. 917-930, ISSN: 0377-2217, DOI: 10.1016/j.ejor.2017.07.061, http://orbit.dtu.dk/files/126231483/TechRep_FischettiPisinger.pdf (visited on 07/07/2018).

Fischetti, Matteo

1995 *Lezioni di ricerca operativa.* Edizioni Libreria Progetto Padova.

Kruskal, Joseph B

1956 "On the shortest spanning subtree of a graph and the traveling salesman problem", *Proceedings of the American Mathematical society*, 7, 1, pp. 48-50.

Lodi, Andrea and Andrea Tramontani

2013 "Performance variability in mixed-integer programming", in *Theory Driven by Influential Applications*, INFORMS, pp. 1-12, https://pubsonline.informs.org/doi/pdf/10.1287/educ.2013.0112 (visited on 07/09/2018).

Nedlin, Jakob

2017 *Ant-based Algorithms for the Wind Farm Cable Layout Problem*, PhD thesis, Informatics Institute, https://i11www.iti.kit.edu/_media/teaching/theses/ba-nedlin-17.pdf (visited on 07/07/2018).