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 add some informations 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 **wfcp** research, we used a library of real-life instances that are made 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 estimated the maximum number of connections to the substation, namely the input parameter C. The **??** table contains all the informations.

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 we used for tests.

1.2 INSTANCE TEST BED

The set of tests that we had the possibility to use are the combinations between the 5 instances of wind farms and the 5 sets of real cables. For simplicity and in order to decrease the possibility to make mistakes we adopted

the same a numeration as the **wfcp** 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.

This problem complexity if the cable routing problem is strongly NP-hard according to [cite pdf].They have proved that the problem is NP-hard in two formulation. 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 necessarly 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 beacuse 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)

[è corretta questa introduzione di K e n?]

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

Definition of $y_{i,j}$:

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

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

Definition of $x_{i,j}^k$:

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

Definition of $f_{i,j}$:

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

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:

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

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

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

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

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

[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 1 image shows an example of a graphical solution to the cable routing problem.

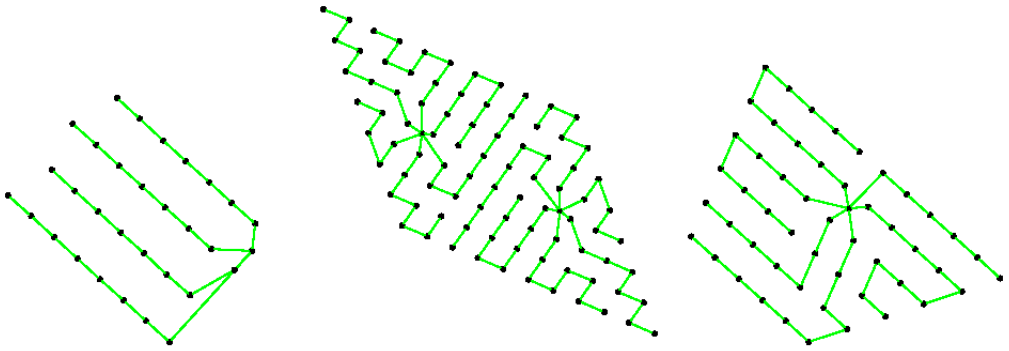

Figure 1: Example solutions to a cable routing problem

2.2 CROSSING CABLES

According to **wfcp**, 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 risk 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; to do this in a calculator environment avoiding mistakes 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, depends on the instance: we noticed the power of CPLEX that find the optimal solution in few seconds, and in the meantime we discovered that some instances takes many hours to be solved by our machines. The main steps of our code in this phase are:

- Read the input files, 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 improving of the solution or to make CPLEX take long time before to find a feasible solution. To avoid those situation it is possible to *"relax"* some constraints in order to make faster the process of searching the first solution. The main steps are to add a slack variable $\geqslant 0$ in the model and then add this variable also in the objective function multiplied for a constant reasonably large. In this way, even if CPLEX could find a wrong initial solution, it's probable that this solution will rapidly get better and became correct. Anyway the choice to relax some constraint should not impact on the optimal solution because of the very high 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 (for some heuristic algorithm) the need to have one big connected component that connects all the arcs.
The first constraint can be relaxed in this way:

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

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

The second constraint, about the flux losses, can be relaxed in this way:

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

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

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:

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

3.1.2 CPLEX Heuristics-Params

The CPLEX code contains some heuristic procedure; being integrated into branch & cut, they can speed the final proof of optimality, or they can 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, changing the frequency of activation of these procedure by setting some CPLEX parameter. We mainly analized 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 don't 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 a too early usage of this method that can lead to a waste of time an performances. We decided to not change this parameter in our tests.

3.2 LAZY CONSTRAINTS METHOD

The generated constraints are often too much and risks to block the solution for a long time if we add them to the model statically. Instead of add systematically all the constraints in the model at the beginning, we generates 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 tecnique 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 cames his name.
The model that we initially use doesn't include the no-crossing constraints and we can't set all of them in 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 cables that have been chosen do not intersect.
This method is mathematically correct, but it seems really unefficient. 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 cames from the fact that often CPLEX spends a lot of time demonstrating the optimality of a solution but maybe this solution uses some crossing cables. The stopping conditions can be several: we can stop at the first solution (not so good in out case because often it is the "star solution"), we can stop when the gap is less than a fixed percentual or we can stop after a timelimit.
In our solution we have choosed a variant of the timelimit condition. We have defined a *timestart* and a *timeloop* parameters: when the algorithm starts CPLEX runs for *timestart* seconds searching 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 uses CPLEX as a *"black box"*, solving consecutively more complex models. It is important to note that CPLEX has a feature that allows to start from the last solution found, it no new constraint is added to the model. One limitation of this method is that at the end it is possible (specially 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 find the optimal integer solution without crossing.
Different from the loop method, CPLEX generate 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 respect to the loop method, hence, in some cases, leading to a slower execution.

3.5 RESULTS

Instance	CPX basic method (TL 5m)		CPX lazy constraint (TL 5m)		CPX lazy callback (TL 5m)		CPX lazy callback (TL 5m)	
	time	solution	time	solution	time	solution	time	solution
data_01	300	2.45E+07	300	2.05E+07	297	1.98E+07	TL	TL
data_02	nan	nan	300	2.16E+07	300	4.02E+09	TL	TL
data_03	300	2.58E+07	300	7.02E+09	300	2.35E+07	TL	TL
data_04	301	3.14E+07	300	2.48E+07	295	2.48E+07	TL	TL
data_05	300	1.00E+08	300	7.02E+09	300	2.40E+07	60	7.00E+10
data_06	nan	nan	300	4.02E+09	300	3.02E+09	TL	TL
data_07	3	8.56E+06	4	8.56E+06			4	8.56E+06
data_08	8	8.81E+06	9	8.81E+06	192	8.81E+06	7	8.81E+06
data_09	9	1.01E+07	31	1.01E+07	3	1.01E+07	2	1.01E+07
data_10	6	1.03E+07	8	1.03E+07	24	1.03E+07	10	1.03E+07
data_12	13	8.60E+06	2	8.60E+06			3	8.60E+06
data_13	7	8.93E+06	5	8.93E+06	9	8.93E+06	4	8.93E+06
data_14	16	1.02E+07	26	1.02E+07	12	1.02E+07	3	1.02E+07
data_15	14	1.03E+07	7	1.03E+07	34	1.03E+07	5	1.03E+07
data_16	225	8.05E+06	20	8.05E+06	30	8.05E+06	21	8.05E+06
data_17	88	8.56E+06	142	8.56E+06	300	8.56E+06	60	8.56E+06
data_18	300	8.36E+06	140	8.36E+06	300	8.36E+06	60	8.36E+06
data_19	102	9.18E+06	161	9.18E+06	46	9.21E+06	60	9.27E+06
data_20	300	1.99E+08	300	1.20E+10	300	1.30E+10	60	4.50E+10
data_21	300	1.17E+08	300	8.04E+09	117	1.50E+10	60	3.10E+10
data_26	300	2.34E+07	300	7.03E+09	300	2.29E+07	60	9.00E+10
data_27					299	2.39E+07	TL	TL
data_28	nan	nan	300	1.70E+10	204	1.20E+10	TL	TL
data_29	nan	nan	300	1.00E+11	2	9.30E+10	TL	TL

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. Its main idea is to use a black-box solver to whom give the input data to generate quickly a first solution. Once the initial solution is 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 are 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 improvement of a solution.
We choose to fix all the arcs with probability 0.5 Using the hard fixing technique, and in general fixing some variables, CPLEX became more 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 has already 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". 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 very likely to choose fixed cables far away from the optimal solution. To avoid this situation ww 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

hard fix normale e dove c'è la condizione della scelta di un ramo, c'è anche la condizione che quel ramo sia presente in entrambe le soluzioni. Abbiamo fatto così perchè altrimenti con due soluzioni simili andava subito a terminare. "fare hard fix sotto le condizioni del RINS"

4.2 SOFT FIXING

This method, also called *local branching*, given a solution called y^{REF}, fixes at least a percentual of the arcs of that solution, and repeat the execution searching the best choice for the others. A critical issue of variable fixing methods is related on 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 good feasible solutions.
A possible implementation is, give 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 the increment is by 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* exploring different solutions type. We decided the specific numbers after some simple test on some instances.
Soft fixing avoids a too 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. [inserire immagine grafico local brancing???]

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. [??? aggiungere .. bo qualcosa sull'integrality grip?]

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, improved incumbent; in practice it compares 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 our specific practice case is more important which are the selected arcs than which are not selected.

4.3 RESULTS

Instance	Hard Fixing (TL 10m)		Hard Fixing RINS (TL 10m)		Soft Asym. Fixing (TL 10m)		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	599	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			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			239	8.60E+06	56	8.60E+06
data_13	19	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			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		
data_26	558	2.28E+07	579	2.24E+07	479	2.24E+07		
data_27	479	2.26E+07	360	2.38E+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 so, many times, when the number of nodes is too high, we can't 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 can compute, relying on the characteristic structure of the problem, a good solution quickly but 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 consecutively a better solution.

5.1 DIJKSTRA

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, it was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later. We use the variant that fixes a single node as the "source" node and finds the shortest path from the source node to all other nodes in the graph producing the shortest-path tree.
In the Dijkstra algorithm initially all the nodes are marked with distance infinite; then step by step we explore all the neighbor and (in lower than the actual) we update the distance. At the end we use the shortest edges to connect all the nodes.
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 in flipping a coin and if it comes out head choose the best of the 10 edge choices, otherwise picks randomly one 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 head this algorithm will 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 execute iteratively all of these steps, 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.

5.3 TABOO 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*, that means that they can not be involved again in the future moves. This type of memorization, in general, is called *recency-based memory* that means that it keeps track of solutions 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 stores, and it directly influences the diversification stage. This parameter is called *tabu tenure* and it is really difficult to decide a-priori its size for two reasons: if the *tabu tenure* is too high the risk is to "freeze" the algorithm, in the sense that only few moves are allowed, and, conversely, if it is too low is not assure that the algorithm could escape from all the local optimums. So, the choice of *tabu tenure* has to be tested in some runs of the algorithm depending on the instance that has to be solved.(cosa abbiamo scelto noi??)
(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 follow what the ants do when them find food. An ant moves randomly in the world leaving pheromones, if it does not find anything continue on its path and the pheromone decay over time. When it finds food, take it and return to its anthill. It retrace the path that has done to arrive at the food doubling the normal quantity of pheromones. The more pheromones on the path make higher the chance that a new ant follows this path. When another ant follows this path reinforced the line of pheromones. With the same logic we can create algorithm 1 for our problem. Initially every edges 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 time of pheromones. Then the algorithm repeat the choice of solution basing of the new probability. Doing so, after some repetition of the loop the probability to choice an edges of the optimal solution should be greater than the other edges.

There are a lot of way to implements the logic of the algorithm as we have find in **nedlin2017ant**, every method for our problem is based on how is implemented findPath function. We chosen to implement the ant colony algorithm based on the Kruscal's algorithm.
The Kruscal algorithm (bibliografia???) is meant to find the minimum coat-

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

ing tree, the tree that link all nodes and that has the sum of the weight of its edges as minimum. The main idea is order the edges with increasing weight and choice the new edge to add at the tree that one that has the minimum cost and does not make cycle with the edges already selected.
We realized the Ant algorithm in Kruscal approach: as a basis we use again the common ACO algorithm before presented but with a modified FindPath function. In this approach we selected the new edge to add at the tree choosing randomly a node and get with the probability given by the pheromones an out edge from the node chosen. The out 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

Instance	Tabu Search (TL 10m)		GRASP (TL 10m)		Ant Algorithm (TL 10m)	
	time	solution	time	solution	time	solution
data_01	600	2.22E+07	600	2.24E+08	600	2.24E+08
data_02	600	2.42E+07	600	2.62E+07	600	2.62E+07
data_03	600	2.65E+07	600	2.85E+07	600	2.85E+07
data_04	600	2.96E+07	600	3.20E+07	600	3.20E+07
data_05	600	2.73E+07	600	2.30E+08	600	2.30E+08
data_06	600	2.71E+07	600	2.97E+07	600	2.97E+07
data_07	600	9.53E+06			600	
data_08	600	9.31E+06			600	
data_09	600	1.10E+07	600	2.13E+08	600	2.13E+08
data_10			600	1.22E+07	600	1.22E+07
data_12	600	8.93E+06	600	1.00E+07	600	1.00E+07
data_13	600	9.65E+06			600	
data_14	600	1.07E+07			time	
data_15	600	1.08E+07	600	1.22E+07	600	1.22E+07
data_16	600	8.55E+06	600	9.07E+06	600	9.07E+06
data_17	600	9.15E+06	600	9.59E+06	600	9.59E+06
data_18	600	9.27E+06	600	9.68E+06	600	9.68E+06
data_19			600	1.11E+07	600	1.11E+07
data_20	600	2.03E+11	600	2.02E+11	600	2.02E+11
data_21	600	2.03E+11	600	2.01E+11	600	2.01E+11
data_26	600	2.43E+07	600	2.48E+07		2.48E+07
data_27	600	2.55E+07	600	2.57E+07	600	2.57E+07
data_28	600	2.01E+11	600	2.01E+11	600	2.01E+11
data_29	600	2.01E+11	600	2.01E+11	600	2.01E+11

Appendices

A | CPLEX

Questo l'ho messo solo perchè nella tesina dei tuoi amici c'è un'appendice su CPLEX .. dacci un occhio e decidiamo se vogliamo qualcosa di simile o no.

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 1 -CC 10 -time_limit 600 -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** : Cross Constraints
  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 | PERFORMANCE PROFILE

E | PERFORMANCE VARIABILITY AND RANDOM SEED

un'altra appendice che potremmo mettere è quella su gnuplot .. noi l'abbiamo utilizzato in qualche modo particolare rispetto ad altri gruppi o abbiamo fatto come tutti?