Università degli Studi di Padova

Operations Research 2: Final Report

Luca Venir (1139089)
Matteo Stringher (1151875)

Contents

1 Problem and MIP model introduction

The goals of this report are the following:

- To describe the "Wind Farm" problem, which was introduced, modeled and solved in class
- To describe the heuristics we have chosen to implement for the project
- To describe the programming environment we have chosen
- To provide the step-by-step installation and configuration guide of such environment
- To introduce and describe the code we wrote during the last semester
- To show and comment the achieved results

In section 1, we introduce every aspect of the optimization problem: we describe its mathematical model, a Mixed Integer Linear Programming model (from now on, MILP or MIP), used to obtain the optimal solution. We also address some first issues, by using some slack variables and by introducing the crossing problem. In section 2 we introduce some heuristics to address this problem; besides the math-heuristics approach, we also chose to implement a genetic algorithm (and everything that follows).
In section 3, we introduce the programming environment we have chosen and the dataset we have received on input; a short guide to the installation of the environment (CPLEX, *Python* and an IDE) over *Linux* can be found in appendix A. Moreover, we describe the implementation choices and we justify them: in this context, efficiency also means better efficacy; we complete our description with the code of the core parts of the program, since *Python* deserves a better clarification.
In section 4, we show the achieved results with some graphs; we will focus on the performance profiling needed to decide which approach is, if it exist, the best one.
In section 5, we draw some conclusions, based on the results achieved and on what we expected to happen.
Appendix A contains as well as the guide to install the tools, a useful script in Python to use the Blade cluster of our department.

1.1 Problem description

Our context is the energy production through wind farms.
A wind farm is a set of wind turbines, placed in certain, strategic, positions. Even though it is possible to have wind farms on the mainland, in this project we are focusing onto offshore wind farms, to be placed in an area near Copenhagen (North Sea, Baltic Sea). In such scenario, placing the turbines (and the cables) is even more expensive than it would be on

land, because of intrinsic costs and due to the need for a certain number of substations on the mainland, which receive all the energy produced offshore. For this very reason, we aim at minimizing the total costs of deploying such wind farms. In particular, given that the wind turbines and the substations are placed (fixed) inside an offshore wind farm, we are asked to optimize its cable routing.

The cable routing optimization problem is the following. Given a set of turbines (with their positions and their energy outputs), a set of cables (with their available quantities, their maximum energy capacity and their cost per unit length) and the substation(s), we want to find an arborescence (more precisely, an anti-arborescence) of cables such that it minimizes the costs given by the cable displacement. Since every turbine produces a fixed amount of energy to be transported to the substation, and such energy flows from one turbine to the next one (until it reaches a substation), the cables to be displaced must be able to bear such energy load. Some less expensive cables are able to hold a small energy flow, while some other cables are able to carry more energy, but they cost more.

In order to describe the turbines in the wind farm, we are given a ".turb" file, which is a plain text file containing, on each row, the (x, y) coordinates of the turbine and its normalized energy production. If the energy production of a row is set to -1, it means that such row represents a substation (which, as just described, receives energy instead of producing it). In order to describe the cables, instead, we are given a ".cbl" file, which is a plain text file containing, on each row, the energy capacity that such cable can carry, its cost-per-unit length and its available quantity.

1.2 The main MIP Model

In order to solve the problem, we use an MILP (or MIP) model proposed by Fischetti and Pisinger [8], whose variables and constraints are described below.
The following first set of variables shows whether or not the i-th turbine has been directly connected to the j-th turbine:

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

The following set of variables will instead tell which cable has been used to connect the i-th turbine to the j-th one.

$$x_{ij}^k = \begin{cases} 1, & \text{if the } k\text{-th cable is placed onto arc } (i,j) \\ 0 & \text{otherwise} \end{cases}, \ \forall i, j \in \{1, ..., n\}, \forall k \in \{1, ..., \text{K}\}$$

And finally, the following set of variables will tell us how much current is flowing between two connected devices:

$$f_{ij} \geq 0, \text{ energy flowing on arc } (i,j), \ \forall i, j \in \{1, ..., n\}.$$

Now, since self-loops are not admitted, we add the first n constraints to the model:

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

It would not make sense to use a cable to connect a turbine with itself. The same set of constraints has been put for f variables.

A second set of constraints must be introduced to add consistency between the chosen y_{ij} and x_{ij}^k variables, because it does not make sense to add more than one cable on the (i,j) arc (when we choose to connect those two turbines). Furthermore, if the (i,j) arc is not selected, no cables must be placed between them. We obtain:

$$y_{ij} = \sum_{k=1}^{\mathrm{K}} x_{ij}^k, \forall i, j \in \{1, ..., n\}. \tag{2}$$

Notice how the n^2 constraints above also imply that $x_{ii}^k = 0, \forall i \in \{1, ..., n\}, \forall k \in \{1, ..., \kappa\}$, guaranteeing no self-loops even on the x variables.

The third set concerns the so-called "out-degree constraints", which is the main requirement for a graph to be an anti-arborescence: For every node, we must have only one outgoing arc if it is a turbine, and no outgoing arcs at all if it is a substation. Namely:

$$\sum_{j=1}^{n} y_{hj} = \begin{cases} 1, & \text{if } p_h \geq 0 \\ 0, & \text{if } p_h = -1 \end{cases}, \forall h \in \{1, ..., n\} \tag{3}$$

Notice that p_h is an input constant; it is either $= -1$ or ≥ 0, depending on role of h.

The fourth set of constraints concerns the substation(s), which have a fixed amount of incoming arcs. Indeed, they cannot accept more than C cables:

$$\sum_{i=1}^{n} y_{ih} \leq C, \forall h \in \{1, ..., n\} \mid p_h = -1. \tag{4}$$

The fifth and sixth sets of constraints concern the energy flow inside the anti-arborescence. The energy given by the turbines must be positive and the energy balance must be satisfied: All the energy outgoing a turbine must be equal to the ingoing energy plus its own production. Namely:

$$f_{ij} \geq 0, \forall i, j \in \{1, ..., n\}. \tag{5}$$

$$\sum_{j=1}^{n} f_{hj} = \sum_{i=1}^{n} f_{ih} + p_h, \forall h \in \{1, ..., n\} \mid p_h \geq 0. \tag{6}$$

The seventh and last set of constraints must be defined in order to guarantee that our cables are able to withstand the energy load onto them. If a cable k is chosen to cover the

(i,j) arc, then its capacity must be at least greater or equal than the energy flow passing by it. We obtain:

$$\sum_{k=1}^{\mathrm{K}} c_k \cdot x_{ij}^k \geq f_{ij},\ \forall i,j \in \{1,...,n\}, \tag{7}$$

where c_k is the energy capacity that cable k is able to support.

Now that every constraint has been correctly defined, it is time to introduce the objective function, which is a cost to be minimized. As we previously asserted in subsection 1.1, we want to minimize the costs given by the cable displacement. Since we are aware of the cost-per-meter of each cable type (k) and of the (x,y) position of each turbine, we can easily compute the total cost with the following:

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{\mathrm{K}} \text{COST-PER-METER}(k) \cdot \text{EUCLIDEAN-DISTANCE}(v_i, v_j) \cdot x_{ij}^k \tag{8}$$

where both functions COST-PER-METER(k) and EUCLIDEAN-DISTANCE(v_i, v_j) return a scalar, that depends on the fixed cable types and on the fixed turbine distances (indeed, v_i and v_j are the points corresponding to the i-th and j-th node). Therefore, this objective function is linear. From now on, for brevity purposes, we will refer to those two functions with the labels COST(k) and DIST(i,j).
We can summarize our mathematical model with the following linear program:

$$\text{Minimize: } \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{\mathrm{K}} \mathrm{COST}(k) \cdot \mathrm{DIST}(i,j) \cdot x_{ij}^k$$

$$\text{Subject to: } \begin{cases} y_{ii} = 0, \forall i \in \{1,...,n\} \\ y_{ij} = \sum_{k=1}^{\mathrm{K}} x_{ij}^k, \forall i,j \in \{1,...,n\} \\ \sum_{j=1}^{n} y_{hj} = \begin{cases} 1, & \text{if } p_h \geq 0 \\ 0, & \text{if } p_h = -1 \end{cases}, \forall h \in \{1,...,n\} \\ \sum_{i=1}^{n} y_{ih} \leq C, \forall h \in \{1,...,n\} \mid p_h = -1 \\ f_{ij} \geq 0, \forall i,j \in \{1,...,n\} \\ \sum_{j=1}^{n} f_{hj} = \sum_{i=1}^{n} f_{ih} + p_h, \forall h \in \{1,...,n\} \mid p_h \geq 0 \\ \sum_{k=1}^{\mathrm{K}} c_k \cdot x_{ij}^k \geq f_{ij}, \forall i,j \in \{1,...,n\} \end{cases}$$

The model we have shown is linear (MILP), with $O(n^2)$ constraints.

1.3 Slack variables

So far the model we have shown correctly describes the problem to be solved. However, when it comes to practice (i.e., putting such variables and constraints into a solver, like

CPLEX), it can be handy to modify the constraints in a way that those solvers can give almost immediately an *incumbent* (that is an integer, feasible solution) even if such solution does not make sense at first or it has a way too large cost. One may say that this approach is not useful at all, or that it is not correct (allowing, e.g., for disconnected turbines, too many cables into the substations, etc.). As we will see shortly, such approach won't affect the final solution returned by the solver, which will respect most of the times the proper constraints and does not affect the optimality of the returned solutions.

What we have just introduced is called *Big-M method*, and it consists of adding a slack variable into some constraints, by relaxing them into a *soft* version of the problem: this means that such constraints admit a violation (described by the value of such slack variable), but also that such violation implies a prohibitive cost in the objective function. In our application, we have introduced two different sets of constraints which can be *relaxed* into a *soft* version. Note how, this change does not affect the total number of constraints, that is still the same: we are indeed *modifying* some constraints into their *soft* version.

Substations constraint. One could relax the constraint described by equation 4 with the following:

$$\sum_{i=1}^{n} y_{ih} \leq C + s_h,\ s_h \geq 0 \forall h \in \{1, ..., n\} \mid p_h = -1, \tag{9}$$

where s_h represents the number of extra (unallowed) cables connected to the substation. The first solution returned by a solver could be a star, connecting each turbine into the substation directly, with a very large objective function value. As we just said, such number of extra connections must be taken into account with a penalty cost into the objective function, which will be modified as follows:

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{\mathrm{K}} \mathrm{COST}(k) \cdot \mathrm{DIST}(i,j) \cdot x_{ij}^{k} + \sum_{h:p_h=-1} M \cdot s_h,$$

where $M \gg 0$ is a very large penalty (e.g. $M = 10^9$ or $M = 10^{10}$)

Flow constraint. An alternative to the previous approach is the following relaxation, which replaces the constraints described by equation 6:

$$\sum_{j=1}^{n} f_{hj} = \sum_{i=1}^{n} f_{ih} + p_h - s_h, s_h \geq 0, \forall h \in \{1, ..., n\} \mid p_h \geq 0 \tag{10}$$

In this case, s_h represents some kind of "flow loss" between turbines: The first solution returned by a solver could be a disconnected anti-arborescence (with a very large objective function). Moreover constraint number (3) must be modified to let some components to be disconnected. The operator will be no more equal to one, instead less or equal to one. The

modified objective function is almost identical to the previous one (the M value is the same):

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{\mathrm{K}}\mathrm{COST}(k)\cdot\mathrm{DIST}(i,j)\cdot x_{ij}^{k}+\sum_{h:p_h\geq 0}M\cdot s_h.$$

This being said, one could choose between (9) and (10) indifferently. In our application, we have found out that the performance of one or the other depends on the instance to be considered and other factors (such as the random seed of the solver). For this reason such performance has been profiled later on (see section 4.3), using all the possible combinations: no slack variables, substation slack variables only, flow slack variables only, and both slack variables.

1.4 Crossings arcs

Before we go any further, and by knowing that solving a linear integer program is *NP-hard*, is it convenient to solve our problem with a linear program approach? *How difficult* is this problem? Is there a different approach that could perform better? We don't exactly know, but we are sure that the problem we have described in subsection 1.1 is *NP-hard*. The demonstration is given by Fischetti and Pisinger [8]. This means that, if $P \neq NP$, there is no algorithm that can solve this problem in polynomial time. As a direct result, our approach (linear programming) can be a good way to face our problem; moreover, the problem size is not prohibitively large, considering that we have instances with at most 100 turbines and $O(n^2)$ non-dense constraints. This does not mean that our problem is easy, but that we neglected a substantial part of it. Indeed, it turns out that what we have described until now is an "*alpha* version" (or easy version) of the problem.
What we did not mention until now is that any admissible solution is an anti-arborescence that *has no crossings within it*; this happens because, in practice, the company in charge of the cable displacement would suffer large costs for each cable crossing with another: the high-voltage underwater would damage them and sometimes it is just not possible to arrange them in that way. Since this problem cannot be ignored (or fixed/arranged by hand), it is necessary to add a whole class of constraints (which we will refer to *no-crossing-constraints* from now on) to the MILP model.

What is a crossing? It is clear that two cables are crossing whenever they are placed one on top of the other. What we will show here is a mathematical way (and therefore algorithmic) to determine whenever two cables are crossing or they are not.

Mathematically, two cables are represented by their ends, p_1, p_2 for the first one and p_3, p_4 for the second one, which have $(x_i, y_i), i = 1, 2, 3, 4$ coordinates; each cable is a segment with

its two ends. The two segments can, then, be parametrized as follows:

$$\text{First segment:} \quad \begin{bmatrix} x \\ y \end{bmatrix}_1 = \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + \lambda \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \end{bmatrix}$$

$$\text{Second segment:} \quad \begin{bmatrix} x \\ y \end{bmatrix}_2 = \begin{bmatrix} x_3 \\ y_3 \end{bmatrix} + \mu \begin{bmatrix} x_4 - x_3 \\ y_4 - y_3 \end{bmatrix}$$

So, as linear algebra says, these two segments cross each other *if and only if* $\begin{bmatrix} x \\ y \end{bmatrix}_1 = \begin{bmatrix} x \\ y \end{bmatrix}_2$.

This means that we have to solve such 2x2 system, where we know every $(x_i, y_i), \forall i = 1, 2, 3, 4$, and we want to find λ and μ, which values determine if those two segments are crossing each other or not. Shortly, using Cramer's rule, it is easy to find such solutions; since there is no need to further investigate the resolution of a linear system, we quickly describe the algorithm that computes such system:

- if the determinant of the system is equal to zero, the two lines on which the two segments rest are parallel, and therefore we can't consider that a crossing;
- if the determinant of the system is not equal to zero and $\lambda = 0 \ \vee \ \lambda = 1 \ \vee \ \mu = 0 \ \vee \ \mu = 1$, the two segments are touching each other on some end: this is not a crossing, because the two segments are making the anti-arborescence, and that is what we would like them to do, in the end;
- if the determinant of the system is not equal to zero and $\lambda > 1 \ \vee \ \mu > 1$ the two lines on which the two segments rest are incident, but the segments are displaced in a way that they are too far from each other: therefore, they won't cross;
- otherwise, if the determinant is not zero and if $\lambda \in]0, 1[\ \wedge \ \mu \in]0, 1[$, the two segments are crossing: note how this is the only case in which the crossing happen;

For brevity, from now on, we will refer to a crossing with the symbol $\cap$ (e.g. $[p_1, p_2] \cap [p_3, p_4]$ means that the two segments are crossing each other).

Crossing constraints. The crossing definition we wrote allows us to write the corresponding linear constraints. A first, primitive, version of such constraints is the following:

$$y_{ab} + y_{cd} \leq 1, \forall (a, b, c, d) : [p_a, p_b] \cap [p_c, p_d].$$

Such constraints say that: For each pair of crossing arcs, only one of them can be selected. But there is a catch: in this way we have written *a lot* of very *weak* constraints. We say "weak" (or that they are not too deep) because each cut in the linear model excludes only a small amount of infeasible solutions. We say "a lot" because it is most likely that they

will produce $\Omega(n^4)$ cuts in the model (which is way too much, even for small values of n). Our aim is to add less but deeper constraints. One way to decrease such a large number of constraints is to avoid repetitions, by choosing (a, b, c, d) carefully: despite that, we would decrease the constraints by a constant amount, but they would still be $O(n^4)$ constraints. This is not enough.

One smarter way to decrease their number is to write the following, based on the fact that there cannot be loops in an arborescence:

$$\forall (a, b, c, d) : [p_a, p_b] \cap [p_c, p_d],\ y_{ab} + y_{ba} + y_{cd} + y_{dc} \leq 1,$$

which is a way stronger constraint (i.e. we generate better cuts), since this constraint can cut away a lot of non-integer crossing solutions. For example, $y_{ab} + y_{cd} \leq 1$ would not get violated if $y_{ab} = 0.5$ and $y_{cd} = 0.5$, while instead $y_{ab} + y_{cd} + y_{ba} + y_{dc} \leq 1$ would be violated with $y_{ab} = 0.5$, $y_{ba} = 0.1$, $y_{cd} = 0.5$ and $y_{dc} = 0.3$. Still, these are $O(n^4)$ constraints.

A better version. In [8], better (i.e., fewer and deeper) constraints have been proposed. Consider the following set:

$$Q_{(a,b,c)} = \{(a, b), (b, a)\} \cup \{(c, d) \in \delta^+(c) : [p_c, p_d] \cap [p_a, p_b]\},$$

where $\delta^+(c)$ is the set of outgoing arches from the point c. The *no-crossing-constraints* can be expressed as it follows:

$$, \sum_{(i,j) \in Q_{(a,b,c)}} y_{ij} \leq 1, \quad \forall (a, b, c)\, s.t.\, |Q_{(a,b,c)}| \geq 3.$$

These constraints impose that, fixed a triple (a, b, c), if a (a, b) segment is chosen, then we won't choose (b, a) and neither we will choose every outgoing segment from c that crosses (a, b), since the solution to be returned is an anti-arborescence.
Therefore, the weak $O(n^4)$ constraints will be replaced with these $O(n^3)$ equations, which are way stronger. Again, we can skip some repetitions by carefully choosing (a, b, c) (e.g. once (a, b) is fixed, do not pick (b, a) and do not pick $c = a \vee c = b$).

Lazy Constraints, Loop Method, Callbacks. Despite the effort made until now, there are instances of our problem where $O(n^3)$ constraints are way too many. For example, even if we are smart and we avoid some repetitions, with $n = 50$ turbines we could add up to $62'500$ constraints, which is better than $6'250'000$, but it is still too much to deal with: recall that solving an integer linear problem is an *NP-Hard* problem. This means that our solver may be so slow that it would not ever find the optimal solution (or it won't demonstrate its optimality).

For this reason, it is unlikely that these constraints will be added as "hard" constraints to the model: as we will see later on in section 4.3.4, it is possible not to add these constraints

right at the start, but instead we can add them before updating the incumbent solution (*Callbacks*), or by adding them on the fly after some computation time (*Loop Method*). All these methods have their strengths and drawbacks, and there is no "recipe-for-all" that works for every problem and instance; indeed, there is no guarantee that adding every constraint to the model since the beginning will achieve bad results on our instances; see section 4.
In the remaining of this section we describe the loop method at a high level, found in algorithm 1, while the other methods will be described in 3, as they are more implementation-oriented.

Algorithm 1 Loop method

Input: The instance's model
Output: Solution

```
Build the model
Run the solver with a short time limit
initial_gap = current gap from the solver
while (Solution is not optimal or Solution has crossings) and (time hasn't run out) do
    Run the solver for a fixed amount of time
    Get the crossing edges in the returned incumbent, say L
    if |L| > 0 then
        Add the corresponding violating constraints
    else
        Save this solution as the best feasible incumbent found (if better than the previous one)
    end if
end while
```

2 Heuristics

In this section we will describe the two main methods used to find fast, sub-optimal solutions.

2.1 Matheuristics

Prior to 2000, it was common to "open" CPLEX and provide some heuristics to ease the computation. Nowadays it is useful to use CPLEX as a black box and work on the mathematical model to enhance the results. Matheuristics are heuristic algorithms made by the interoperation of metaheuristics and Mathematic Programming (MP) techniques. An essential feature is the exploitation in some part of the algorithms of features derived from the mathematical model of the problems of interest [3]. A definition of metaheuristic will be given in section 2.2. Matheuristic methods let to constraint the search-space. In such a way the optimum is not reached (or it is not possible to prove the optimality) but a very good solution is given in a shorter time. In the following two techniques are presented: hard and soft fixing.

2.1.1 Hard fixing

The hard fixing is an example of a matheuristic. In this technique CPLEX is not modified internally, instead the model is modified. The pseudocode is given in Algorithm 2. The step number 5 is not necessary, but is useful to ease the computation in the no-crossing version. In fact, we do not want a crossing solution. If one edge is fixed to be chosen, all the edges crossing can not be included in the solution.

Algorithm 2 Hard fixing

Input: The instance's model

Output: Sub-optimal solution

1: Build the model and run the solver
2: **while** Solution is not optimal OR time limit is not expired **do**
3: Select a number of edges from the last solution
4: Temporary fix

$$y_{ij} = 1 \quad \forall \quad \text{selected arc}$$

5: Set to 0 all the y_{ij} variables that cross a selected arc
6: Solve the new model and remove the temporary fixings
7: **end while**

This matheuristic allows to find a sub-optimal solution in a reasonable amount of time. In fact results have shown that using this method, the optimum with some edges fixed, can be reached much faster than with normal techniques.

2.1.2 RINS

RINS, which stands for Relaxation Induced Neighborhood Search, is an example of a similar technique. RINS has been proposed by Danna et al. in 2005. The idea behind RINS is quite easy. At each branch_and_bound node the integer solution can be compared with the fractional solution. If two values coincide, fix that variable to the integer value. From their paper:

- Fix the variables that have the same values in the incumbent and in the current continuous relaxation
- Set an objective cutoff based on the objective value of the current incumbent
- Solve a sub-MIP on the remaining variables for a limited period of time

In CPLEX this routine can be invoked with a frequency given as parameter to the model. This algorithm is another example of matheuristic. The model is modified through an heuristic, adding some constraints. Our model benefits of such tuning and the analysis is shown in section 4.3.

2.1.3 Soft fixing

As the name suggests this is a softer version: Arcs are not forced to be equal to 1 or 0, given a reference solution y^{RIF}. Instead, the new constraint forces to fix a percentage of the arcs to y^{RIF} solution, e.g. use 50% of these arcs. This technique has been prosed by Fischetti and Lodi [10]. This approach does not force to find the solution in the neighborhood. The reduction explained in hard fixing can not be applied to soft fixing, since we do not know which edges will be fixed. While hard fixing may get stuck in a local optimum, soft fixing can often move on his search. Two versions are proposed.

- **Symmetric**
  Instead of fixing the variables as shown in algorithm 2, a different constraint is added to the model.
  $$\sum_{(i,j)|y_{ij}^{RIF}=0} y_{ij} + \sum_{(i,j)|y_{ij}^{RIF}=1} (1 - y_{ij}) \leq K \tag{11}$$
  The parameter K sets the upper bound of flips that are allowed.
- **Asymmetric**
  In some problems the number of 0's in the solution is much higher than the number of 1's. This leads to a very dense constraint with a lot of variables. This might slow down the overall computation. This is what happens in our problem, the constraint is then reduced. In the asymmetric version only flips from 1 to 0 are included.
  $$\sum_{(i,j)|y_{ij}^{RIF}=1} y_{ij} \geq (n-1) - K \tag{12}$$

This means: Remove only a certain amount of edges from the y^{RIF} solution.

2.2 Metaheuristics introduction

The methods described before, modify the mathematical model to induce a local search. At the end of the course this approach has been left out to analyze alternative ways to solve the Wind Farm problem. CPLEX and the mathematical model will not be used anymore.

The cable routing problem is an NP-Hard problem, and therefore some heuristics must be applied to provide a solution in a reasonable amount of time. The search-space of candidate solutions grows exponentially as the size of the problem increases. An exhaustive search for the optimal solution is infeasible even for medium-size instances, since they do not apply to all the problems. Heuristics are problem-dependent techniques. For example, the use of Kruskal's algorithm in this context is an heuristic: It would produce a minimum spanning tree (MST), but there is no guarantee that an optimal MST is an optimal solution for our problem (as we will show in the result, it surely is not). In the remainder of this paragraph a definition of this concept is given and our approaches implemented are described.

A meta-heuristic is an algorithm designed to solve approximately a wide range of hard optimization problems, without having to deeply adapt it to each problem. Indeed, the Greek prefix 'meta' is used to indicate that these algorithms are 'higher level' heuristics, in contrast with problem-specific heuristics. Meta-heuristics are generally applied to problems for which there is no satisfactory problem-specific algorithm to solve them [4]. Indeed, in the worst scenario, a problem could not be even represented by a mathematical model, and therefore in this case a meta-heuristic is not just useful, but needed. Luckily, the Wind Farm problem allows to build a mathematical model, allowing us to compare the two approaches. Indeed, the MIP model gives a lower bound on the quality of the solution. This leads to a rough analysis of the solution cost.

When a solution is sought using heuristics, the process is divided in two sub-processes. In the first, called constructive phase, a meta-heuristic approach is implemented to solve the specific problem (and therefore the algorithm becomes, necessarily, problem-dependent, an heuristic); this approach produces some bad and rough solutions, which are given as input to the second phase, called repair phase. In the Wind Farm problem a modified Kruskal's or Prim's can be used. Another approach would exploit the BFS (Breadth-First search): one could choose to set the substation as the starting vertex, and then begin a randomized breadth-first exploration; the final output, instead of a simple visit, is the anti-arborescence (more details will be given in section 2.3.3). The constructive phase may produce solutions which are some orders of magnitude far for from the optimum. After building one good solution or a set of solutions, the real meta-heuristic is applied to 'refine' them. Many algorithms have been proposed in the literature, starting from simulated annealing in 1982, or the well known Tabu Search, which is implemented in the LocalSolver software. In 1998,

Koza registered his first patent on genetic programming [4]. Another famous algorithm is the "ant colony" algorithm. All of these repair heuristic algorithms share two alternated phases: diversification and intensification. When intensification kicks in, the algorithm greedily searches for something that looks like a very good solution, but with a shortsighted approach. When it comes to diversification, instead the algorithm widely explores the solution space without focusing too much on refinement. A good balance between the two is the key factor: the diversification phase could bring some new "ideas" to the solution found until now, whereas the intensification phase is the *core* of a refinement algorithms. They will be well-described later on. In our project we decided to work on two constructive methods and on genetic algorithms as the repair phase.

2.2.1 Cost function

First of all, it is mandatory to define an objective function. The cost of the solution is no longer provided by CPLEX. The objective function is conceptually the same as before (when considering the big-M method), but its cost evaluation must be implemented by us.

$$\sum_{i=1}^{n}\sum_{j=1}^{n} COST\_MIN\_CABLE(i,j)\cdot DIST(i,j)y_{i,j} + \sum_{i=1|p_h\geq 0}^{n} \mathbb{1}(i)\cdot M_1 + n_s\cdot M_2 + n_c\cdot M_3 \quad (13)$$

The function *COST_MIN_CABLE* assigns at each arc the least-cost cable feasible for the flow on it. The number n_s counts the number of cables linked to the substation over the maximum allowed, which is defined on each instance. Instead, n_c represents the number of crossings in the solution. The indicator function is defined as:

$$\mathbb{1}(i) = \begin{cases} 1, & \text{if the outgoing arc has flux larger than the highest capacity cable} \\ 0, & \text{otherwise} \end{cases} \quad (14)$$

The formula can be modified for efficiency purposes for some repairing meta-heuristics, like the Tabu Search. Furthermore, it is useful to give an higher value to the second term of equation 14, multiplying the penalty by the difference of the flux and the highest capacity cable: For example, if the flow on a cable has value 17 and the highest capacity value is 15, then $(17-15)\cdot M_1$ should be its cost. In such a way a direction for the search is induced by the objective function. As we will show later, this is not needed for genetic algorithms.

2.3 Constructive phase

Now it is time to dive into the details. Our first, constructive phase must build a set of solutions (and not just the single best one). This is not always the case: Refinement techniques such as the Tabu Search require a good, single solution to work on (so it is possible to change it and hopefully improve it). The genetic algorithm, instead, requires to

start with a set of solutions (called population), which are as different as possible (species), even if bad (i.e., even with prohibitive costs).

Three constructive heuristics have been used in the project: GRASP, randomization of costs and randomized BFS. The first two methods adopt a customized algorithm for the MST problem to find the solutions. In fact, by looking at the Wind Farm problem, it can be noticed that some characteristics are shared with MST: A minimal tree must be found with extra constraints. Obviously, it is not sufficient to apply the Kruskal's algorithm to find an optimal solution, otherwise the problem would not be NP-hard.

All the solutions have a given cost according to the aforementioned formula, with the little modification to take into account edges with flux greater than the capacity of the cable with maximum capacity. For some problems we could use the so called multi-start technique. Each run of the algorithm would give a feasible solution with a different cost. This can't happen with the adaptation of Prim's algorithm: the given MST would be the same, even if the algorithm starts from a different vertex.

2.3.1 GRASP

GRASP has been introduced in 1995 by Feo and Resende. The acronym stands for "Greedy Randomized Adaptive Search Procedures". It is an iterative multi-start meta-heuristic approach for difficult combinatorial optimization problems. In our problem we have modified the Prim's algorithm in order to generate a population of randomized solutions. Thinking about it, the problem is now reversed: From an algorithm designed to be optimal (for the MST problem), we need to modify it to generate non-optimal solutions. At each step, Prim's algorithm adds to the tree the edge with minimum distance from the set of vertices already belonging to the tree: The algorithm must be modified in this point. A new set of k-closest vertices is defined at each iteration (where k is a parameter). Then, one of them is randomly sampled to be added to the tree. For a better understanding of the idea behind it, the pseudo-code is shown in Algorithm 3. According to Feo and Resende, RCL is the restricted candidate list, i.e. the set from which a choice is made. An especially appealing characteristic of GRASP is the ease with which it can be implemented. Few parameters need to be set and tuned (i.e. the candidate list size and the number of GRASP iterations). Therefore, the development can focus on implementing efficient data structures to assure quick GRASP iterations.[7] Moreover, it is very easy to parallelize this algorithm. Only one variable is shared among the processes: the best solution found.

2.3.2 Cost randomization

Cost randomization is easier to understand and it does not imply any modification of Prim's algorithm. To put it simply, costs are randomly modified in an interval, according to a uniform distribution, with mean value the true distance. In such a way each run of the

Algorithm 3 Prim's algorithm modified with GRASP technique

Input: The graph $G = (V, E)$ where $c : V \times V \rightarrow d(v_1, v_2)$ with $d(v_1, v_2) = \sqrt{(v_{1,x} - v_{2,x})^2 - (v_{1,y} - v_{2,y})^2}$ and $k = |RCL|$

Output: Sub-optimal solution

1: $T := \emptyset$
2: $S := 0$ // Substation index
3: **while** $|T| \neq n - 1$ **do**
4: Find the k-lightest edges $[v, h] \in \delta(S)$ and $h \notin S$
5: Sample one edge $[v_s, h_s]$
6: $S := S \cup h_s$
7: $T := T \cup [v_s, h_s]$
8: **end while**

Prim's algorithm is computed on a different graph. This method is quite straightforward to be implemented.

2.3.3 Randomized BFS

Until now, we have implemented methods and techniques that exploit some well-known algorithms (such as the Prim-Dijsktra algorithm). The technique we are about to show now does not exploit a meta-heuristic approach or a well-known algorithm. Instead, we want to implement a pure problem-oriented heuristic, so that we can compare its performance with the previous algorithms. Also, as mentioned before, the reason we have showed three different constructive methods is because of the refined algorithm we have chosen (i.e. the genetic algorithm): The more different the solutions, the better the results.

This method explores the complete graph induced by the nodes of the wind farm given; like a BFS would do on such graph, the algorithm visits the adjacent nodes of a starting node chosen, and repeats until there are no more nodes to visit: our algorithm starts from the substation and it is forced to visit the closest nodes first, out of its adjacency list. Indeed, fixed a current node, the algorithm is forced to visit no more than its first i closest adjacent nodes in the complete graph; here, i is a parameter given. For the sake of randomization, at each iteration step, the algorithm actually visits and links j nodes out of the closest i ones. Here, j is another parameter and obviously it must be $j < i$. In other words, at each iteration step and fixed a current visited node, the algorithm selects the i closest nodes in the adjacency list (in the complete graph) of the current node and randomly selects and links j nodes to it (taking care not to create loops). The algorithm considers each node in the visited set (*FIFO*) and stops when every node has been visited (and hopefully linked). Some refinement is needed, though: It may happen that, since some of the i considered nodes may never be selected in the randomization step, some nodes are left behind (i.e. never connected

to the tree); this is solved by forcing the connection between such nodes and its closest node the tree at the end of the procedure .
For the sake of clarity, we show the pseudo-code in Algorithm 4.

Algorithm 4 Constructive method: Modified BFS

Input: The graph $G = (V, E)$, where $c : V \times V \rightarrow d(v_1, v_2)$, with $d(v_1, v_2) = \sqrt{(v_{1,x} - v_{2,x})^2 - (v_{1,y} - v_{2,y})^2}$;
$j, i \in \mathbb{R} : j < i$.

Output: Sub-optimal solution

1: $T := \emptyset$
2: Let $s = 0$ be the substation
3: Select the i closest nodes to s
4: Out of such i nodes, randomly choose j nodes
5: Insert such nodes into a queue of nodes to be visited, say Q
6: Set the substation as visited, by initializing a set of visited nodes $L = \{s\}$
7: **while** $|Q| > 0$ **do**
8: Remove node n from the Q (FIFO policy)
9: Label n as visited, by adding n to L
10: Select the i closest nodes to n
11: Out of such i nodes, randomly choose j nodes, let A be the selected nodes
12: Remove already visited nodes, if any, i.e. $A = A \setminus L$
13: Update the to-visit queue, i.e. $Q = Q \cup A$
14: **end while**
15: **if** $|L| < |V|$ **then**
16: Get every leftover node, i.e. $B = V \ L$
17: For each $v \in B$, connect it to its closest neighbor.
18: **end if**

2.4 Refinement phase

2.4.1 Evaluation function

In order to get more practical, we need to define what *fitness* means in the context of our problem; to do so, we take the evaluation function (see equation 14), which measures the *fitness* of a single individual, and take the mean of the whole population: This is our *fitness* evaluation function. The evaluation function is the first step to create a genetic algorithm. To be more specific: it is the *fundamental* part. It is the function that estimates the success of our specimen: it will allow us to divide the population between non-successful fish and the alpha-male ones. The goal of this separation is that, later, the successful specimen (i.e. better solutions) will have more chance to get picked to form the next generation. As simple

as it appears, this is the core of the genetic algorithm, but it also is the most problematic part. Since such evaluation function must be run on every solution inside the population (which may be made of several hundreds of specimen), its implementation must be the most efficient as possible: the efficacy of these algorithms depends a lot on their efficiency. Their bottleneck is, indeed, the evaluation function, because it is run at least once every generation (and the number of generations can reach the thousands). The evaluation function is shown in algorithm 5.

Algorithm 5 Evaluation function

Input: The solution tree T, the instance constraints (e.g. C), the big-Ms values (e.g. $M_1 = 10^{12}$) and a distance function $d(v_1, v_2) = \sqrt{(v_{1,x} - v_{2,x})^2 - (v_{1,y} - v_{2,y})^2}$.

Output: The cost of the given solution tree.

1: $S := 0$ // the costs to be calculated
2: $S = S + \text{costs\_from\_cables\_length}(T, d, M_1)$
3: **if** There are more than C cables connected on T **then**
4: $\quad e := |\text{cables\_on\_substation}| - C$
5: $\quad S = S + M_2 \cdot e$
6: **end if**
7: crossings := $\emptyset$
8: Examine the edges and locate every crossing
9: $S = S + M_3 \cdot |\text{crossings}|$

2.4.2 Use of the constructive methods

One of the first steps into the implementation of a generic algorithm is the creation of the specimen, or, in other words, the population from which the generations will follow. We have widely covered the constructive methods in subsection 2.2, but the point here is: how many of the GRASP method solutions should we include in the starting population? How many of the BFS method ones? Not so surprisingly, there is no clear answer to this question. In our implementation, we have chosen to parametrize this choice, by dividing the population into sevenths and by searching for the better configuration among the following:

- No constructive methods are used (the population is made only of random trees);
- $\frac{1}{7}$ is made of GRASP solutions and $\frac{1}{7}$ of BFS solutions (so that $\frac{5}{7}$ of the population is composed of random trees);
- $\frac{2}{7}$ is made of GRASP solutions and $\frac{2}{7}$ of BFS solutions (so that $\frac{3}{7}$ of the population is composed of random trees);

- $\frac{3}{7}$ is made of GRASP solutions and $\frac{3}{7}$ of BFS solutions (so that $\frac{1}{7}$ of the population is composed of random trees);

This is a totally arbitrary decision, but if our goal is to diversify the population, *even with ugly, random, unfeasible tree solutions should be used*; as we have said before, this should help the initial diversification rate. The reason why we have chosen to create random trees only in the first configuration, is because we want to see what happens if great starting diversification is given. We show what happens if we do so in the result section, where each method is compared. After all, we expect the constructive phase to be useful (i.e. that lets the genetic algorithm give better solutions).

2.4.3 Encoding a tree into a chromosome

The last function needed to begin the genetic heuristic, is an *encoding* (and, therefore, a *decoding*) algorithm; encoding is needed because the genetic approaches require a *chromosome*, which is a sequence of characters representing (uniquely) a specimen. In our context, this means having a sequence of numbers (nodes) that uniquely represent the solution (tree). The encoding and the decoding functions must be:

- Efficient to the point that their use is transparent to the rest of the algorithm, since the efficiency bottleneck is already in the evaluation phase;
- Bijective, because there should not be any loss of information between the transformations.

***Prüfer* numbers.** Although these two properties make sense, we are aware of just one tree-encoding function which respects them efficiently: the *Prüfer* encoding function. This algorithm guarantees the two aforementioned features and it is very easy to implement and exploit. In our implementation, we have shown the performances of the *Prüfer* method and two more encodings.

Prüfer showed that a lossless encoding of a tree can be achieved with Algorithm 6, with only $n-2$ digits; the resulting number is called *Prüfer number*. However, this encoding can be inconvenient in our context: As we are showing in the next subsection, the cross-over process may not make sense with it.

For completeness, we show the decoding process of the *Prüfer* number in Algorithm 7.

Furthermore, one last remark should be done for the *Prüfer* encoding: This algorithm was designed for undirected trees, which is not the case of our context; one could simply fix this small problem by making simply precautions. Instead of activating a "repair" procedure on the undirected tree (by simply directing every edge towards the substation), we have chosen to improve the encoding by adding one more digit: basically, the encoding procedure is repeated until the tree has one single node left, while the decoding procedure is modified

Algorithm 6 Prüfer encoding

Input: A undirected tree T, made by n vertices.

Output: A lossless tree encoding, made by $n-2$ digits.

$PN := 0$ // Our output
while $|T| \geq n-2$ **do**
  $i = \min \text{leaf}(T)$ // Smallest labeled leaf
  $j = \text{parent}(i)$ // j is incident to i
  Add j as the rightmost digit into PN
  Remove i from the tree
  Remove the $\{i, j\}$ edge from the tree
end while

by adding directed edges instead of undirected ones (i.e., by adding the (i, j) edge at each iteration). This is easier and it saves some computation at the cost of memorizing one more digit.

Algorithm 7 Prüfer decoding

Input: The lossless *Prüfer* encoding PN, made by $n-2$ digits.

Output: The corresponding undirected tree T, made by n vertices.

$T := 0$ // Our output
$D := \{i : 1 \leq i \leq n\} \setminus PN$ // vertices not included in PN
while $|PN| > 0$ **do**
  $i = \min D$ // Smallest labeled vertex in D
  $j = PN[0]$ // Leftmost number in PN
  Add the $\{i, j\}$ to T
  Remove i from D
  Remove j from PN
  if $j \notin PN$ **then**
    Add j to D // If j does not occur again in the encoding, add it to the D set
  end if
end while

Other encodings. As an alternative to the *Prüfer* encoding, we have proposed an encoding that exploits the successor of every node inside the directed tree: Since every node inside T has one parent (besides the root), we encode this information on a encoding (with length n), made of the successors of every node (recall that we are working on an anti-arborescence). For example, if the node labeled by a 4 has 6 as its successor, the encoding will write 6 in the fourth position; the root of the tree will have itself as its successor, so that its position will contain itself.

Given a tree T, this choice gives a "lossless" encoding; however, the opposite is not always true: It easy to construct an encoding that represents a graph with disconnected components and loops (so that the result, T, is not an anti-arborescence anymore). Therefore, this encoding is not lossless in a complete sense. Again, this may be a problem when it comes to the other parts of the algorithm and it will be explained in the next subsection; in fact, the problems emerge when it comes to breeding, generating random chromosomes (first step) and inserting random mutations (see section 2.4.5).

2.4.4 From one generation to the next

Since there is the *Prüfer* encoding, why should someone look for different encodings? Besides the aforementioned problems, it turns out that the encoding has a strong impact on the performance of the breeding phase, which is another core part of the genetic heuristics. Until now, we have not mentioned *how* the genetic algorithm combines two specimen in order to obtain a new one. To do so, it emulates the breeding process found in nature: it takes the two *chromosomes* of two selected specimen and performs a cross-over on them. A cross-over produces a new chromosome, which has some parts from the mother (a solution) and the other ones from the father (another solution).
One could choose several ways to perform the cross-over: For example, the cross-over can be contiguous, by breaking the chromosomes in two (e.g. in half or in a random point) and by interchanging their pieces together and therefore obtaining two new chromosomes of the same length of their parents. In our opinion, this choice would not be optimal; we want an algorithm that, most of the time, keeps the "good" choices made until a certain generation: Instead, what happens with this cross-over is that such choices are easily deleted (or forgotten). Here, "choices" are the good tree edges selected and kept until a certain point: Choosing to arbitrarily change the genes' position inside the chromosome destroys the memory property we are looking for.

Instead, we have chosen to perform the cross-over by sampling (randomly) each gene of the two chromosomes. When the parents share the same gene in the same position, we are certain that their children will be the same in there; instead, whenever mother and father are different, one children inherits the mother's gene, while the other, the father's. This choice ensures that the following generation always keeps the good genes of the previous parents, while offering some sort of diversification, or "choice": As it was previously stated, a genetic algorithm decides which specimen should survive by checking their solutions' cost.

Picking the parents. One could choose how to pick the breeders (say, the parents). There are lots of way to do this, but there are two main ideas to keep in mind: the goals are to select the best solutions of the previous generation and not to *completely* put aside the others. For example, by selecting only the good solutions since the very beginning of the genetic algorithm, the risk is to converge really quickly towards a local minimum and

not towards the best solution possible. Our solution is to set our genetic algorithm into the *intensification* phase until *it looks like* we are stuck into a local minimum for several generations. Here, every new generation is formed by randomly picking the breeders (and not just the best ones); but since the population number should not be growing exponentially (instead, it is fixed to a certain limit), intensification occur by *discarding* some solutions at each phase: in the *intensification* phase, we keep only the better ones (actually, we keep a very, very small fraction of bad ones). The *diversification* phase, instead, joins the game by adding way more *bad* solutions into the population and by *mutating* the chromosomes by a lot.
Finally, when breeding, we think that the *alpha* individual deserves to mate with every element in the population; this is not just an analogy with wild life: our choice allows the best chromosome found to transfer its good parts to everyone, while everyone can give this solution some changes that (hopefully) will improve him. We think that this helps the genetic algorithm by a lot.

A first repairing technique. The problem with breeding (and actually with some other parts of the algorithm) is that the encoding chosen can get in the way. For example, when using the *Prüfer* encoding, by applying the aforementioned breeding process (cross-over), the resulting tree may not have conserved the "good" features of its parents (but, on the other hand, it ensures that the new chromosome actually represents a tree); this is the reason why we have chosen to explore different encodings, as we have shown in the previous paragraph. Indeed, our intuition is not only confirmed by the results (see section 4), but it turns out that this hypothesis has been widely explored in literature (see [11]); long story short, Gottlieb et al. say that the use of *Prüfer numbers* cause poor performances in evolutionary algorithms. Therefore, we chose to explore the use of the *successor* data structure. This encoding is useful when it comes to breeding, because it saves what the parents have in common (the *hopefully good* edges), but it creates troubles for the remaining genes: the resulting "successor" data structure (or encoding) may represent a graph with loops and disconnected components. For this reason a well-thought repair procedure must be implemented (and not just because of breeding); such procedure must be as efficient as possible and it should be included in the encoding (i.e. transparent to the other sections of the algorithm).
Implementing a repair procedure inside the decoding function *does not make the encoding lossless*. Indeed, we are forced to make some choices that can not preserve what we have given in output: usually, this is something that is not a tree (i.e. a graph with loops and multiple disconnected components). Given that we see a loop inside our data structure, what options do we have? Before giving our two solutions, let's have a look at the following property.

Property 2.1. *Let:*

- *$G = (V, E)$ be a directed graph such that $\delta^+(v) \leq 1$, $\forall v \in V$*
- *$S \subseteq V$, such that it induces a connected component of G*

Then, the number of cycles in the graph induced by S is less or equal than 1.

This property motivates the repair algorithm we are about to describe.

- First, the algorithm locates every root in the graph: having more than a root means having some disconnected components (recall that a root has no successors, and this should happen one time at most); by our implementation choice, a root is identified when a node has itself as its successor (since we assume that no self-loops are allowed, this greatly simplifies the code).
- Second, it explores every connected component found with a deep-first search, starting from the roots found previously; recall that, because of property 2.1, we can not have some connected components that contain more than one cycle: this means that, if there are some nodes left after such visit, the components left behind have got exactly one cycle. This activates a procedure that fixes the loop by exploring the component: when the loop is found, the last directed edge is removed and the penultimate vertex is set as the new root.
- The first and the second step are iterated until no cycles are found in the graph; at this point, we may have several disconnected components: therefore, the last step consists in connecting every root (for each connected component, besides the substation) to the nearest node (not belonging to its connected component).
- Finally, the tree is eventually re-directed towards the substation, and the repair procedure is finished.

Although this procedure may look very imaginative and dispersive, we have thought of this process because that is what we would do, by hand, with pen and paper; furthermore, we have implemented the above repairing function efficiently, by exploiting some proper data structures.

A more elegant repair technique. Another solution is to exploit a well known MST algorithm, Kruskal's Algorithm. The idea is simple: The repair function assigns a null cost to the edges resulting from the encoding (i.e. the chromosome, also known as the solution), and it appeals to Kruskal's algorithm, which will pick such edges first; if some of them induce a loop, they are discarded (because of how Kruskal's algorithm works) and the edge search goes on. Hopefully, this more elegant solution will always select the most convenient edge (where the *convenience*, here, is measured with *distances only*).
One could say that the first repairing technique is useless, compared to the second one,

because Kruskal's algorithm will always ensure an MST out of the given chromosome. This is not necessarily true for two main reasons. First, this problem is way harder than a mere MST search: As we argued before, an MST often leads to a non-feasible solution. Second, the implementation of Kruskal's algorithm is inherently less efficient than the first technique. As we argued before, being inefficient implies having bad results (efficacy) in the same amount of time. We will argue more about this in the results section.

2.4.5 Mutation

The last step of the genetic algorithm makes an analogy with gamma rays: the natural but random mutation of an individual. The idea is that, right after the breeding phase, each individual has a small probability to see their DNA change a little bit: the goal of this operation is to prevent the algorithm to be blocked in a *local minimum*. Local minimums are what heuristics are all about: with a small window of time, the goal is to find a "good" local minimum, without being stuck into a "bad" one. As it is mentioned at the beginning of this section, this motivates the introduction of a *intensification* and a *diversification* phase. In the genetic algorithm context, intensification means sharpen the population in order to find a good solution (i.e. discarding bad chromosomes only); instead, in the diversification phase, mutations occur often and the mutation probability is high.
The mutation phase introduces some more uncertainty: what is a "good" mutation probability? And: at which point should we introduce mutations? As we've already learned with heuristics, there is no clear answer; furthermore, it is easy to see that this genetic approach introduces a lot of parameters: our choices are oriented to minimize the amount of parameters introduced. Indeed, in the intensification phase we (almost) do not introduce mutations, except for the breeding phase for when duplicates occur (i.e. a single gene mutation for each duplicate chromosome); in the diversification phase, we expect the population to mutate much more.

3 Implementation and Programming Environment

Before starting to work, we have selected and chosen the tools to develop the project. As MIP solver we could either choose CPLEX or Gurobi. The former was used during the project. As programming language used to develop our tools, we have chosen Python: The use of CPLEX's Python API is slightly different from other programming languages, and therefore this section is focused on showing such peculiarities (besides showing some core parts of the actual Python implementation). Furthermore, here we show the most meaningful implementation choices. Some listings are shown and described to compare the syntax in the libraries developed by IBM.

3.1 Environment overview

CPLEX is developed by IBM and has years of experience. The CPLEX Interactive Optimizer is an executable program that can read a problem interactively or from files in certain standard formats, solve the problem, and deliver the solution interactively or into text files. The CPLEX Callable Library is a C library that allows the programmer to embed CPLEX optimizers in applications written in C, Visual Basic, FORTRAN, or any other language that can call C functions. The engine is written in C, but many languages are supported through an API: *C++*, *Java*, *.NET* and, finally, *Python*. The CPLEX connector for The MathWorks MATLAB enables a user to define optimization problems and solve them within MATLAB using either the MATLAB Toolbox or a CPLEX class in the MATLAB language.[12]

Python's CPLEX API. CPLEX offers two libraries to interact with the optimizer: the Python API and DOcplex. The former is a full-featured Python application programming interface supporting all aspects of CPLEX optimization. The latter has a easier to use syntax and is *NumPy*-friendly, but it is limited. We have chosen to try both to test features and limits of this language in the CPLEX software.
Our judgment about the integration of CPLEX in Python will be given in section 5.

IDE and documentation. Documentation has been written using *Sphinx*. This tool is similar to Javadoc for Java and it allows to create and format documentation.
PyCharm is the Integrated Development Environment (IDE) we have chosen; it has been developed specifically for the Python language and it is distributed by JetBrains. Thanks to his usability developing in Python is easier. PyCharm Professional Edition is free for open source projects and for some educational uses. We have used the free Academic license for both PyCharm and CPLEX.

Latex and Version Control LaTeXhas been used to write the report, and we have written it with *Overleaf*. Overleaf is an on-line free LaTeXwriter and offers a live cloud service for

easy-sharing. It supports many packages. For example, the *Minted* package has been used to include and color listings.
As version control system, GIT has been adopted and we jave chosen Github as our hosting service. A public link to our code is available.[1]

3.2 Python overview

Python is an interpreted language with many capabilities, which ease the development process. Among its strengths the one most appreciated is simplicity. In the Zen of Python one aphorism is: *simple is better than complex*. Take a look at the list.[2] Nevertheless, it doesn't offer the same performance as other languages; indeed, performance is its main drawback. Some of its useful features have been used many times during the project: list comprehension, functional programming, dictionary, sets.

3.2.1 Imported Libraries

Python has a wide range of hundreds easy-to-use libraries. Here are listed the ones that we have used and that are the most important ones to our project.

- *cplex*: The way to access the fully-featured Python library
- *docplex*: The API, developed by IBM, already mentioned.
- *plotly*: Rising plot library
- *matplotlib*: Plotting library
- *networkx*: Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex graphs and networks.
- *warnings*: Handling warnings in Python
- *time*: Time-related functions
- *argparse*: Easy handling and parsing of command line arguments
- *math*: Used for basic math operators
- *collections*: Lets to use named tuples
- *paramiko*: Implementation of the SSHv2 protocol
- *heapq*: Heap-related operations library
- *pprint*: Literally "pretty-print", a useful printing library,s for a easy debugging

[1] `https://github.com/strin-matey/OR2`
[2] `https://www.python.org/dev/peps/pep-0020/`

3.2.2 Object-Oriented Programming with Python

While the main concepts are quite similar to the other languages, Python has some differences when it comes to OOP (and they've been deeply exploited in this project): For example, information hiding in Python is complete, while *get* and *set* methods are implemented with very elegant overrides called *properties*. *Properties* are so useful (and they are the core difference in OOP with Python) that they deserve a proper discussion.

A *property* is an attribute to be given to functions, in order to define a new name associated to the class: properties can be widely used for multiple purposes, but they are mostly used to implement an override of the dot operator (attribute call) and of the assignment operator (when modifying an attribute by assignment); respectively, these two overrides replace the usual *get* and *set* methods implemented in other languages, such as Java. For instance, in general, when we create a private attribute, we're supposed to create those two *get* and *set* methods to access it from outside; we can invoke and exploit them by writing those method names (e.g. OBJECT.OURATT_GET()). Python, instead, offers the possibility to add some sort of transparency about the information hiding of a particular attribute: it allows to retrieve an attribute by simply calling its name, without calling any method (e.g. object.ourAtt), since information hiding is handled somewhere else. For example, say that, at first, we want OURATT to be public; then, in Python, we would create it inside the constructor, as it follows:

```
class OurClass:
    def __init__(self, a):
        self.ourAtt = a


x = OurClass(10)
print(x.ourAtt)
```

If we act like this, we don't need *get* and *set* methods; indeed, we decide to print OURATT without calling any function (that would be absurd and overabundant). Now, say that, after writing the code, we discover that (for some reason) OURATT must be a value between 0 and 1000: How can we fix the situation? If we were using a, say, "old" language, we would have to re-write most of our code, because every time we need to call such attribute we would have to invoke use the old *get* and *set* methods. Instead, Python's properties allow us to avoid that, by simply adding the information hiding property of OURATT separately:

```
class OurClass:
    def __init__(self, a):
        self.ourAtt = a


    @property
```

```
    def ourAtt(self):
        return self.__ourAtt

    @ourAtt.setter
    def ourAtt(self, val):
        if val < 0:
            self.__ourAtt = 0
        elif val > 1000:
            self.__ourAtt = 1000
        else:
            self.__ourAtt = val

x = OurClass(10)
print(x.ourAtt)
```

Notice how the *use* the attribute OURATT in the main script has not changed, while the *get* property takes care of both the override of the dot operator and the information hiding (by simply adding a new syntax operator, the *at* symbol); the *set* property, instead, does proper *set* methods operations (in our example, we choose not to raise *Exceptions*). This small example has been used to show what is the *pythonic* way of programming (e.g. the attribute SELF must be used everytime we are defining a class method), while showing how *get* and *set* methods has been implemented in our project.

3.3 Project organization

Now that the main Python features have been covered, we describe the most important parts of the implementation that differs from a classical C-oriented implementation; we also describe the core parts of the implementation that are necessary to understand the structure of our code.

3.3.1 Variables' indices

As it is known, CPLEX counts every variable added to the model by simply counting their entrance number; such number is called *variable index*. Since we're aware of the order in which variables are ordered, but we want to make the code independent from such order, we use a function that, given a variable, it returns its index. As an example, we report here one of such functions (here, we show the x one).

```
def __xpos(self, i_off, j_off, k_off):
        """
        Get the position of the column inside the model
```

```
        :param i_off: First turbine number
        :param j_off: Second turbine number
        :param k_off: Cable index

        :return: Column index
        """

        return self.__x_start + i_off*self.__n_nodes*self.__n_cables +
        ↪  j_off*self.__n_cables + k_off
```

The formula used in the RETURN statement holds because, whenever we add variables, we save the *starting* position of the x variables (while the number of nodes and cables are well known). For example, if we're wondering if the solver has placed a cable of type $k = 2$ (out of three cable types) between the substation ($i = 0$) and the fourth turbine ($j = 4$), this function would return its index; if we assume that we have $n = 81$ nodes and that the x variables were added first (so that X_START $= 0$), the index returned would be $0+0*81*3+4*81+2 = 326$: with such index it is possible to query CPLEX to find such variable value. We have found this function very useful in the whole project, and not just for debugging purposes.

3.3.2 Reading the input

Reading a file or parsing the command line is quite straightforward in Python. By exploiting the *argparse* library, there's no need to implement an accurate (i.e. character by character) parsing function. We decided to store our input in a list of named tuples.
Let's take a glimpse to the method reading the turbines file.

```
def read_turbine_file(self):
  # the following opens and closes the file within the block
  with open(self.turb_file, "r") as fp:
    for line in fp:
      words = list(map(int, line.split()))
      self.__points.append(self.Point(words[0], words[1], words[2]))
  self.__n_nodes = len(self.__points)
```

In such a way, points will be easy to read: for example, we can get the x coordinate of the first point with: `__self.points[0].x`.
The cables file is read in a very similar way, obtaining a list of "Cables" *namedtuples*.

Parsing the command line. Parsing the command line is also straightforward; here, we show a short example on how to add a single argument parse.

```
parser = argparse.ArgumentParser(
  description='Process details about instance and interface.'
)
parser.add_argument(
  '--crossings',
  choices=['no', 'lazy', 'loop', 'callback'],
  help='Choose how you want to address the crossing problem'
)
args = parser.parse_args()
if args.crossings:
  self.__cross_mode = args.crossings
```

We find this example self-explanatory: This library allows to add strict constraints to the argument parsed and the parsing is started with a simple function invocation. The object ARGS will contain the parameter passed as an attribute, which name corresponds to the argument name given; this is possible because, in Python, attributes can be added to objects on-the-fly; at this point we handle the input by checking its value or simply by storing it in our instance object, WINDFARM.

3.4 Model building

Once the model has been described (in section 1), it wouldn't be necessary to show the details of the implementation in the report, since they can be found in the code in our repository; while these details are more or the less the same in languages such as C, C++ or Java, it is worth mentioning the exploitation of lists comprehension in Python, used to build the model with the API. Furthermore, CPLEX and DOCplex have two different behaviors, and this deserves a comparison.

3.4.1 CPLEX usage

In this subsection we describe how we chose to build the model described in section 1 with the *CPLEX*'s *Python* APIs. First, the variables should be added. Here, as an example, we show how the x_{ij}^ks variables are added.

```
# Add x(i,j,k) variables
self.__x_start = self.__model.variables.get_num()
self.__model.variables.add(
  types=[self.__model.variables.type.binary]
        * (self.__n_nodes**2)
        * self.__n_cables,
  names=["x({0},{1},{2})".format(i+1, j+1, k+1)
```

```
        for i in range(self.__n_nodes)
        for j in range(self.__n_nodes)
        for k in range(self.__n_cables)],
  obj=[cable.price * WindFarm.get_distance(v,u)
        for v in self.__points
        for u in self.__points
        for cable in self.__cables]
)
```

As we can observe, the *add* function demands three parameters: *types*, *names* and *obj*, which require a list of variable types, a list of variable names and, eventually, a list of objective function corresponding coefficients, respectively. Notice how the variables are loaded in the *Pythonic* way (with a list comprehension) and how the code is very much readable and bug-free.
As an example, here we show how a slack variable should be added: they are simple, normal variables (nothing more, nothing less).

```
if self.__flux_slack:
  # Add slack variables for flux
  self.__flux_slack_start = self.__model.variables.get_num()
  self.__model.variables.add(
    types=[self.__model.variables.type.continuous]
          * self.__n_nodes,
    names=["s2({0})".format(h + 1)
          for h in range(self.__n_nodes)],
    obj=[1e9] * self.__n_nodes,
    ub=[max([cable.capacity for cable in self.__cables])]
        * self.__n_nodes,
    lb=[0] * self.__n_nodes
  )
```

Here, what we wanted to show is the possibility to add a lower bound (as well as a upper bound), directly at the variable definition; also, notice how the star operator is overridden for lists, in Python: It creates a list of objects in the leftmost operand, with length equals to the rightmost operand. Similar overrides are present in Python (e.g. the plus operator, that performs a concatenation).
Next, we want to show how we added our constraints into the model, using *CPLEX*'s API.

```
# Flow balancing constraint
self.__model.linear_constraints.add(
  lin_expr=[cplex.SparsePair(
```

```
        ind=[self.__fpos(h,j) for j in range(self.__n_nodes) if h!=j] +
            [self.__fpos(j,h) for j in range(self.__n_nodes) if j!=h] +
            [self.__flux_slackpos(h)] if self.__flux_slack else [],
        val=[1] * (self.__n_nodes - 1) + [-1] * (self.__n_nodes - 1) +
            [1] if self.__flux_slack else []
    )
      for h, point in enumerate(self.__points)
      if point.power >= 0.5
    ],
    senses=["E"] * self.__n_turbines,
    rhs=[point.power for point in self.__points if point.power > 0.5]
)
```

As it is shown above, the *add* function asks for three parameters: *ind*, which is a list of specific objects called *SparsePair*, *senses*, which specify the inequalities' senses as a list (for each constraint added) and *rhs*, which are the right-hand-side of the constraints to be added. In other words, *CPLEX* forces the user to re-write every constraint in a way such that the linear combination of the variables (indeed, the *SparsePair*) must be on the left, while the single and simplified constant value must be on the right (indeed, the *rhs*).
To define a *SparsePair* (i.e. the summation of variables on the left-hand-side), *CPLEX* requires two parameters: *ind*, which is the list of variables to be added (via variables' names or indexes), and *val*, which are the corresponding coefficients, for every variable added. There is a one to one match between the variables inside the *ind* list and the *val* one: a mismatch between these two lists will cause *CPLEX* to raise an error; furthermore, notice how we have chosen to address the variables with their indexes, by calling the *pos* function described in section 3.3.1, rather than calling them via their name. Indeed, we have found out that it is possible to extract variables from CPLEX via their names: this is way more readable and practical, but it has a heavy performance cost; for this reason, we have adopted the index solution.
Finally, notice how *Python* allows us to write the slack variables inside this constraint if and only if we have chosen to add them; also, notice the readability of such choice, thanks to the *Pythonic* shorthand syntax.

3.4.2 DOcplex usage

Another API, distributed by IBM, can be used to solve the model: DOcplex; its uniqueness deserves a concise explanation. To avoid any misunderstanding, we will refer to the library described before as the Python API, while the other library we are going to described will be called DOcplex.
The DOcplex API ensures better code readability, since it gives the possibility to use an

easier semantics (which are completely different from what we have described until now). Here we define the *DOcplex* model with a certain time limit.

```
def build_model_docplex(self):
  # Create the model
  model = Model(name=self.name)
  model.set_time_limit(self.time_limit)
```

Variables' names in the model are always inserted using +1 to match the mathematical model. Here we present an example on how such variables are added in *DOcplex*.

```
# Add y(i,j) variables
self.__y_start = self.__model.get_statistics().number_of_variables
self.__model.binary_var_list(
  ((i+1, j+1)
  for i in range(self.__n_nodes)
  for j in range(self.__n_nodes)),
  name="y%s"
)
```

Notice how the Pythonic way of programming simplifies the variables input and prevents bugs from happening in this context. Just like we would do with the *CPLEX* API (see subsection 3.4.1), we are able to get such variables by their names or by their indexes. As previously argued, it is better to do it with their indexes for the sake of efficiency.
The function *add_constraints* brings the most differences in the semantics/syntax between *DOcplex* and *CPLEX*. Here we bring, as an example, the constraints described by equations 4 and 6: we first create a list of all the variables at the left-hand side of such inequality (with a *Pythonic* loop) which are, then, reduced with the sum function, as the equations say; we repeat the process whenever is necessary. The constraint is completed by using common mathematical semantics, by writing the inequality as we would do on paper (kind of). Notice how we're not bound to write variables-only on the left hand side and a single number on the right hand side; also, there are no restraints on the variables' repetition, as opposed in *CPLEX*. Here follows the example.

```
# Flow balancing constraint
self.__model.add_constraints(
  self.__model.sum(
    self.__model.get_var_by_index(self.__fpos(h,j))
    for j in range(self.__n_nodes)
  )
  ==
  self.__model.sum(
```

```
    self.__model.get_var_by_index(self.__fpos(j,h))
    for j in range(self.__n_nodes)
  ) + self.__points[h].power

  for h,point in enumerate(self.__points)
  if point.power > 0.5
)
```

Notice how in the second constraint call we ignore slack variables for the sake of simplicity, but it would have been easy to add them with another *Pythonic* loop (we actually do it in the code). Lastly, the way to express the objective function is different in *DOcplex*, since it doesn't allow to set the coefficients in the objective function while defining the variables (as opposed of *CPLEX*); in DOcplex, it makes sense to write down the objective function as we would do on paper: by writing down the summation to minimize.

```
# Objective function
self.__model.minimize(
  self.__model.sum(
    cable.price *
    WindFarm.get_distance(u, v) *
    self.__model.get_var_by_index(self.__xpos(i, j, k))

    for k, cable in enumerate(self.__cables)
    for i, u in enumerate(self.__points)
    for j, v in enumerate(self.__points)
  )
)
```

Notice how the *sum* reduce function perfectly interacts with the *Pythonic* lists comprehensions.

3.4.3 Comparison between the libraries

During the project we did not have any issue with the full Python API. It offers a new layer above the API coded in C. In such a way the difference in performances is negligible. Indeed, the model is built with the Python instructions, but the computation is done using the C code. We had the possibility to use all the features required for the Wind Farm problem, even the advanced ones, like callbacks.

Instead, the DOCplex library offers a more limited set of features, at the moment. For example, it is possible to set the RINS parameter or the polish time parameter, but there is no

way to use callbacks or more advanced features. Anyways, DOcplex still has some commercial advantages. This library can be either used with the well-known CPLEX software or on the IBM Cloud. IBM offers the possibility to solve the model on their cluster with relative low prices, on a pay-per-hour basis.[3] In such a way a full license is not needed.
Since all the CPLEX's capabilities can be used with the student license, we have decided to use and exploit Python API, so that we can avoid any constraint given by the DOcplex library.

3.5 Graphs

Since plotting is a core part of the visualization process in the project, and since we have exploited two popular Python libraries, we're spending some few words to describe them.
The problem has been plotted using two different libraries:

- Plotly
- Matplotlib

Both of them can exploit NetworkX's library to store the data. It can handle the graph data structure. NetworkX is a well known library for plotting in Python. It provides a standard programming interface and graph implementation that is suitable for many applications.
We stored each node of the wind farm, giving a different label to substations and turbines. Then, edges are added to link vertexes using a directed graph. NetworkX uses both Graphviz and Matplotlib to draw the graph. An example is shown in figure 2. Different colors are used for different types of cable.

Plotly is a up-and-coming library that uses NetworkX as data structure. The library lets to plot nice and interactive graphs even when using *Overleaf*. We have tested both. An example is shown in figure 3. Using the interactive mode, by hovering one turbine its number is shown.

3.6 Computation cluster *Blade*

Final considerations about our implementation choices must be done when it comes to the use of the departmental computation cluster, since we have exploited some Python characteristics here as well; here, we show how we quickly launched *jobs* with the support of a script.
To efficiently exploit all the features on the cluster, we have written a Python3 script to be able to upload new files and launch jobs on the cluster without moving *by hand* the files and writing the .job file. Details about the cluster's architecture are given in section 4.1. The script, called *cluster.py*, connects us to the cluster with the SSH protocol. User name and

[3] `https://developer.ibm.com/docloud/documentation/decision-optimization-on-cloud/pricing-plans-and-free-trial/`

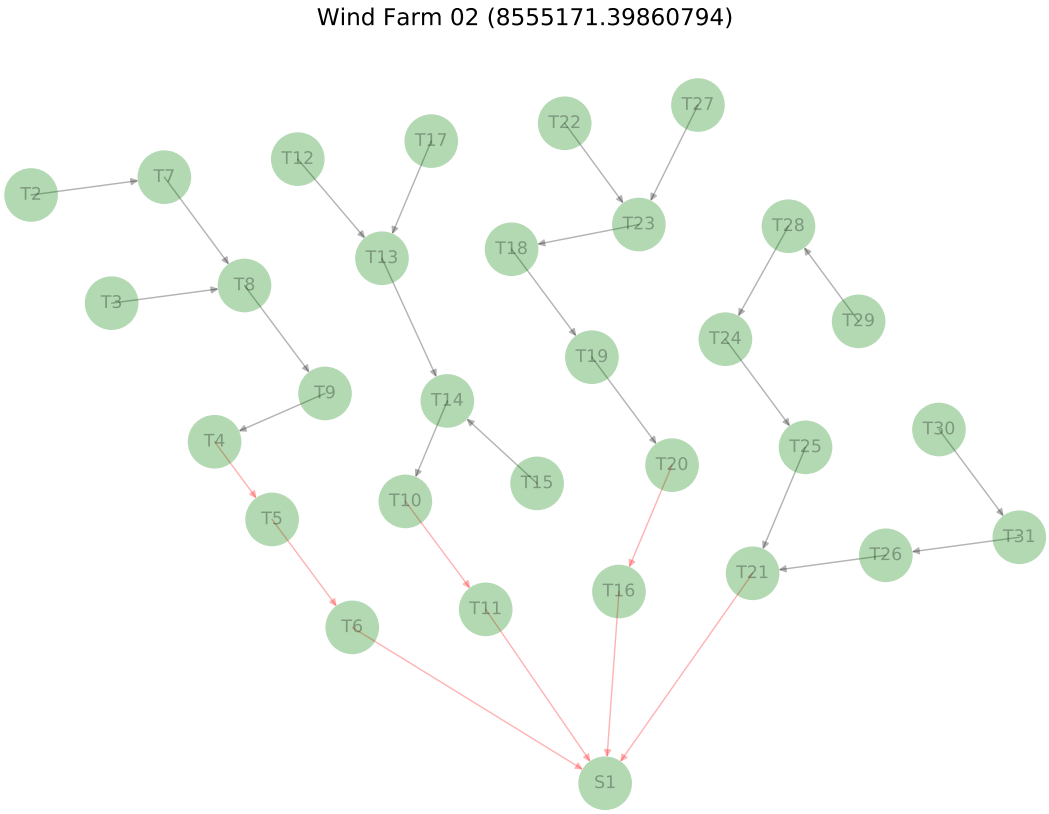


Figure 2: Solution for dataset number 7

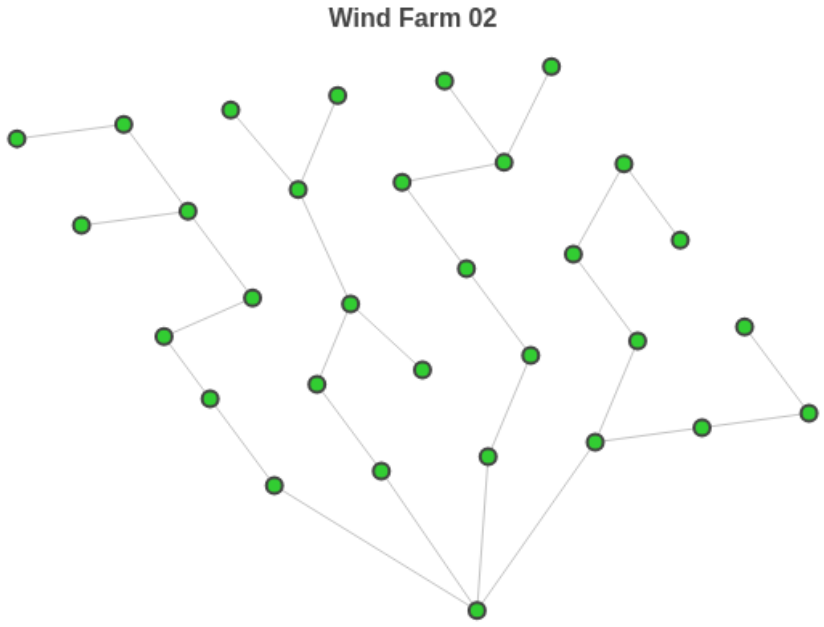


Figure 3: Solution for dataset number 7 with Plotly

password are asked to perform the connection.
The script takes the files, given in a list, and transfer all of them to the proper folder in the cluster. In such a way it is easy to upload new edited files. Some instructions inside the script file are used to build the *commands.job* file. This file is uploaded to a newly created folder with the timestamp as name. An example for the analysis of RINS options is given below.

```
# All the options for the job
dataset_numbers = [1, 2, 3, 4, 5, 6, 20, 21, 26, 27, 28, 29]
interfaces = ['cplex']  # DOcplex is available for some features, too
rins_options = [-1, 0, 10, 100]
num_iterations = 3
files = ['src/__init__.py', 'src/lib/WindFarm.py'] # Files to be transfered
```

The script explores every given possibility; the job is composed of several program launches, one for each parameter combination. At the end of the script, the header for the *results.csv* files is prepared for the performance profiling graphs. During the computation, this file will be filled with the required data. The job is, then, launched inside the script with SSH. With this script all the steps are easier and all the output files are stored in the aforementioned folder.

4 Results

In Operations Research it is important to test different techniques and understand the results given back with different instances and algorithms. In the first half of the course CPLEX and mathematical programming have been exploited to solve the Wind Farm problem. The results are presented and commented in subsection 4.3. In the second half, cheaper methods were used; starting from scratch, we have developed algorithms to solve the same problem (in subsection 4.4), as mentioned in the introduction. The second approach will show if a mathematical solver, like CPLEX is truly needed. The optimal solution might not be reached in practical times in some applications and hence a mathematical solver may be too slow. Sometimes, the problem cannot even be modeled and CPLEX can not be used; luckily, this is not Wind Farm case. Conclusions, comparisons and considerations will be given in section 5.

4.1 The architecture

All the tests and results given in section 4 are computed on the 'Blade' cluster, provided by the department. In such a way results can be compared with other groups. A basic configuration is needed to use CPLEX with Python on the cluster. Instructions on how to do it are provided in appendix A.
The cluster is made out of 14 blades, model DELL PowerEdge M600.[4] Each of them is equipped with:

- 2 quad core Intel Xeon E5450 (12MB Cache, 3.00 GHz)
- 16 GB RAM
- 2 Hard disk of 72GB. RAID-1 (mirroring) is used

The management system used is the Sun Grid Engine. It is a grid computing computer cluster software system (known as a batch-queuing system). It allows to launch jobs on the cluster with the *qsub* command. Grid Engine is typically used on a computer farm or high-performance computing (HPC) cluster and is responsible for accepting, scheduling, dispatching, and managing the remote and distributed execution of large numbers of standalone, parallel or interactive user jobs. It also manages and schedules the allocation of distributed resources such as processors, memory, disk space, and software licenses. SGE has a list of features, among the most important:

- Scheduling algorithms
- Cluster queues

[4] `https://www.dei.unipd.it/bladecluster`

- Job and scheduler fault tolerance
- Parallel jobs (MPI, PVM, OpenMP)

The list is quite long, some of the features were used in the project.

4.2 Timing and technical details

Some details must be written and explained to understand our results.
The cluster is shared among all the students and researchers: In order to avoid problems due to conflicts, we have decided to use CPU time instead of real time. This option can be set inside CPLEX and CPU time can also be retrieved with Python. In such a way we are sure that our results are not modified by preemption by other jobs. In the following, we will always refer in the text and plots to CPU time. Real time is never mentioned.

All the algorithms and procedures have been tested three times on each instance. Some methods, like hard fixing, use randomization, so it is useful to repeat tests many times. Not all the instances have been used, only the "hard" ones. Here are listed.

```
dataset_numbers = [1, 2, 3, 4, 5, 6, 20, 21, 26, 27, 28, 29]
num_iterations = 3 # Number of iterations on each instance
```

In the following, for each plot is specified the timeout set. Some tests needed more time than the others. For example the RINS analysis is lighter than the no-crossing problem. Each job needed at least 12 hours.

Performance profiling has been used to plot the results. This type of plot does not only tell which algorithm is better than the others. Benchmark results are generated by running a solver on a set P of problems and recording information of interest such as the number of function evaluations and the computing time. The performance profile is a means to evaluate and compare the performance of the set of solvers S on a test set P.[6] The plots show which solver (i.e. algorithm or technique) outperforms the others and how much. Usually with this type of plot the amount of time needed to solve the problem is used. In this scenario, it is impossible to solve all the instances in a reasonable amount of time. Instead of time we have plotted the cost of the best solution found within a time interval. A solution with lower cost means a better solution, just as with time.

Moreover, to be sure that no previous solution is used we have set the 'advanced start switch' parameter. For MIP models, setting 1 will cause CPLEX to continue with a partially explored MIP tree if one is available. If tree exploration has not yet begun, setting 1 (one) specifies that CPLEX should use a loaded MIP start, if available. Setting 2 retains the current incumbent (if there is one), re-applies presolve, and starts a new search from a new root.[5] Instead setting it to 0 means do not use advanced start information. We have used

[5] https://www.ibm.com/support/knowledgecenter/SSSA5P_12.8.0/ilog.odms.cplex.help/CPLEX/Parameters/topics/AdvInd.html

the value 0 as default. Instead, when needed, the value 1 has been set to use previous work. It is necessary in hard and soft fixing and with the loop technique.

Another parameter plays an important role in CPLEX: the seed. Before each iteration we have set a different one. Variation in the random seed can increase diversity of results. This problem is called performance variability. CPLEX as default runs in deterministic mode, but a little difference in the very first choices in the branch tree can modify the entire computation. For example a different choice at the root of the tree will affect the following nodes, because decisions in MIP are based on previous results. To achieve a good statistical analysis of the methods we have run each computation with a random seed.

All the computations have been checked before plotting through the error file printed by the cluster.

4.3 Results with CPLEX

Here, we list our experiments with the use of CPLEX. Our objectives include:

- Find the best RINS parameter for our context;
- Make a analysis between the slack constraints used;
- Find the best practice to face the no-crossing problem

4.3.1 RINS

Even if RINS is a matheuristic, due to its importance, we made a standalone analysis.
The parameter can be set before solving the model and can vary CPLEX speed. The parameter is an integer value, that sets the frequency of times RINS is used. The value -1 means do not use RINS, 0 use the default behavior in CPLEX, 10 use RINS every 10 nodes in the branch tree.[6] RINS has some overhead and an high frequency is not useful in every problem.

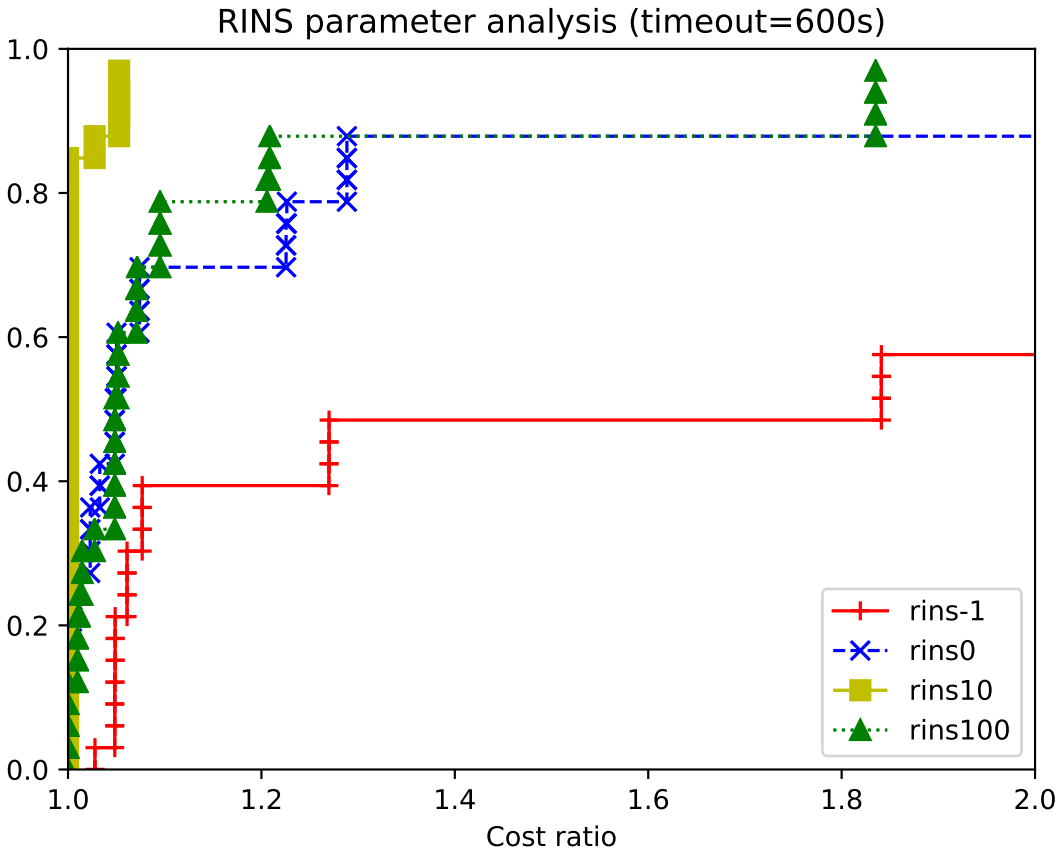
Results are depicted in figure 4. Our analysis showed that RINS is very useful and speeds up the computation. The value 10 has the best results and beats the other values most of the times. Values 0 and 100 have similar results.
RINS was published and then implemented in CPLEX in 2005. It was a true breakthrough in Operations Research. Our results confirm its usefulness in the Wind Farm problem.

4.3.2 Slacks

The slacks explained in section 1.3 have been implemented and tested. The first difference is clear: A solution is found immediately. Another difference that is not seen in the short-time

[6] `https://www.ibm.com/support/knowledgecenter/SSSA5P_12.8.0/ilog.odms.cplex.help/CPLEX/Parameters/topics/RINSHeur.html`

Figure 4: RINS analysis with four different parameters



is the number of integer solutions. The slacks let to extend the search space and in such a way more integer solutions are found. In other words the polytope is larger and the set of allowed solution is larger too.
With dataset number 1 the first solution is given in less than one second and has cost equal to $8.00000e+13$. This does not mean that a better solution is found in a time interval. This should be the aim, but not all the slacks beat the no-slacks model on each instance. Results are shown in figure 5. The slack has been implemented allowing 'flow loss'. This lets to CPLEX to accept solutions that form subtrees not linked to the substation. Most of times it can be seen from the graphs that some turbines are detached from the rest of the tree.
Instead, the substation-slack has a different behavior. The first solution found is a 'star' with all the turbines linked to the substation. Along with the computation fewer turbines are linked to the substation, and start to link each other in a tree.

It is clear that the flux slack outperforms the other methods most of the times, while the substation slack does not perform very well. The latter on some instances gets stuck and is slower. Thanks to this analysis we have decided to use the flux slack in the no-crossing run, since it is quite heavy and needs a speed-up.

4.3.3 Matheuristics

Matheuristics have been previously described and here we describe our results. The procedures have been tested without extra constraints for the no-crossing problem. We expected to see a better behavior by matheuristics within a short timeout. In fact we have launched two jobs with different timeouts to see the difference. In the first plot (figure 6a) CPLEX without any extra constraint is slower most of the times. Instead with a longer timeout the hard fixing technique shows similar results to the 'No matheuristic' method (see figure 6b). In both plots the soft fixing technique outperforms the others.

To avoid bad solutions both hard and soft fixing start to add constraints to the model when a 'good' solution is found, i.e., the solution has a percentage lower than 30%. In hard fixing, the probability of setting an edge is 0.9 in the first iteration, and is decremented at each step of 0.05, until it reaches the value 0.5. Without explaining other details, the soft fixing technique adds at each iteration a deeper constraint, i.e., the search-space is more restricted.

The timeout has been set with two different parameters: the overall wait time and the CPLEX timeout. The former is specified in the plots 6a and 6b. The latter is the timeout set inside CPLEX. Usually in our tests we have set the internal timeout 10 times less than the overall wait time.

Matheuristics play a big role in medium-large instances where an optimal solution is not needed. They trade quality of the solution with time. But in our analysis soft fixing still gets better results with a long timeout.

Figure 5: Slack analysis with 1000 seconds

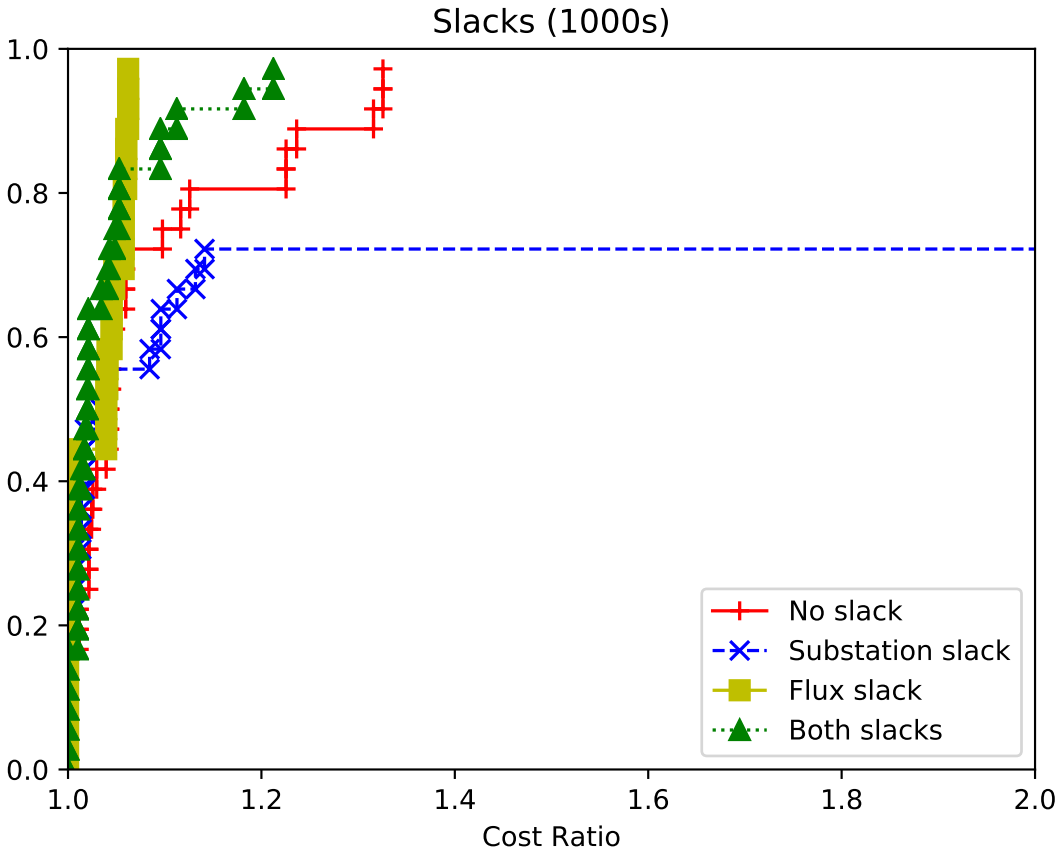
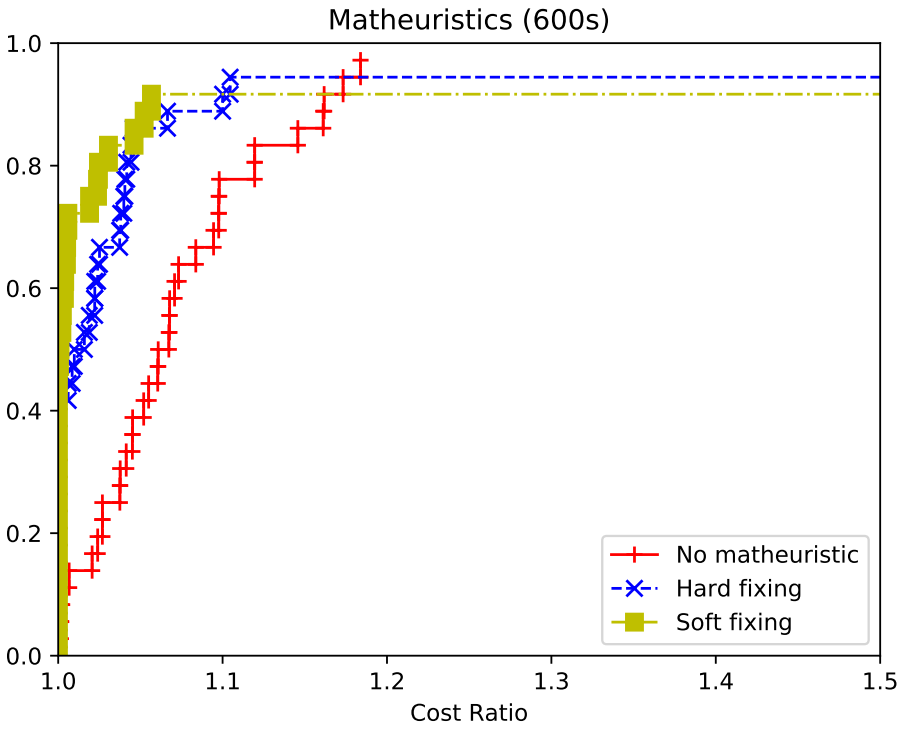
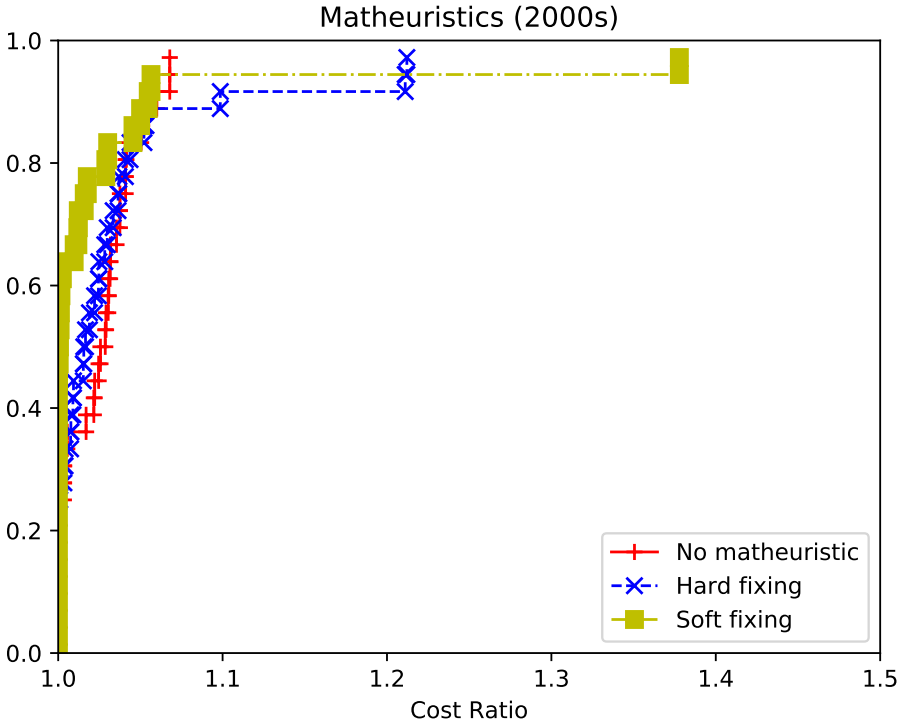


Figure 6: Matheuristic analysis



(a) Matheuristic analysis with 600 seconds



(b) Matheuristic analysis with 2000 seconds

4.3.4 No-crossing solution

Computing the no-crossing problem clearly needs more time. The number of constraints increase hugely in the lazy method, while the loop and callback algorithms do not need to fill the model but they add them when they are violated. In such a way the three methods add the constraints in three different moments.

Lazy adds all the constraint at the beginning. We have subtracted to the timeout in CPLEX, the time spent to add all these constraints. Anyways this does not affect the performances of the Lazy method. In fact in the run, plotted in figure 7a, with 4000s adding the constraint only takes less than 5% of the overall time.

The loop method adds the violated constraints only when the internal timeout in CPLEX is expired. In the same way we have previously set the constraint in hard and soft fixing. The number of constraints is far less, but the computation made before might be lost if none of the solutions is used to start a new branch tree. Furthermore it does not guarantee that the solution found at the end is feasible (i.e. there are some crossing). This depends on the number of iterations of the *solve* method. We have updated the cost of the best solution only when the solution had no crossings, otherwise an high value is set. In table 1 it can be seen that some instances have not been solved in the outer timeout. The value 10^{12} marks this case.

Finally, the callback method adds a lazy constraint before updating the incumbent. When a new solution is found, a procedure set by us checks if there are crossings. If found a new constraint is added to choose just one of the two edges crossing and the new solution is rejected. Instead when a feasible solution is found CPLEX accepts it. This internal procedure must be called many times.

The three methods offer advantages and disadvantages. The results have shown unexpected results. The lazy method seems to be unbeatable! Even if we have subtracted the time spent to build the model, it is faster than the other methods with most of the timeouts. Figures 7a and 7b show that the Callback method is a little bit better than the loop one. But the loop results rely on the timeouts set which are quite difficult to be optimized on each instance. Instead the lazy method is much faster than the others on most of the instances or it is quite close. Moreover table 1 shows that some instances like the 20th are very far from the optimum, while on other instances results are approximately equal to the Lazy method.

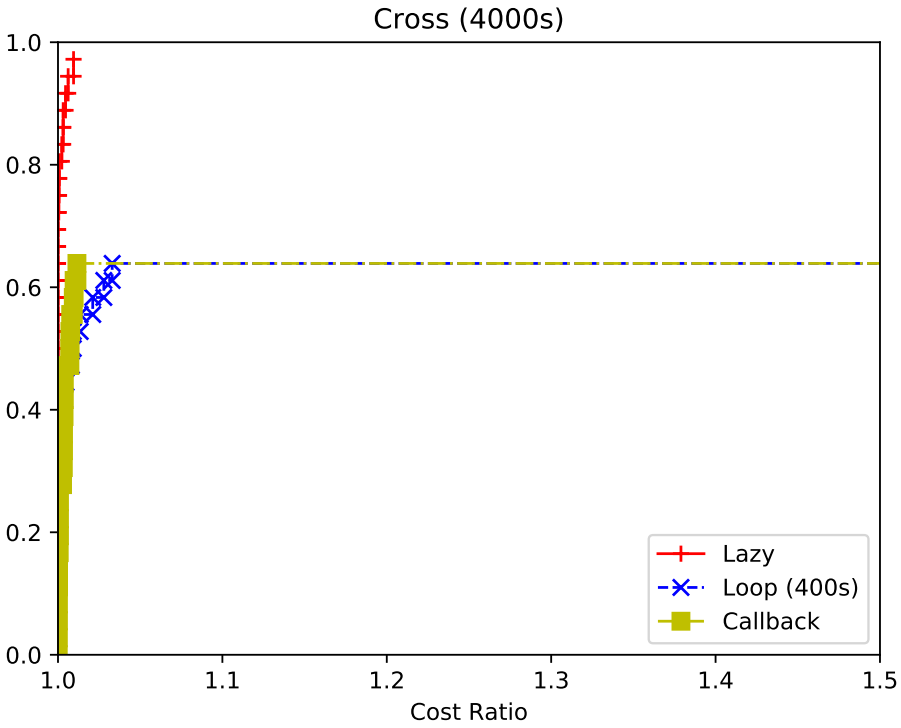
4.4 Results with metaheuristics

Here, we list our experiments without the use of CPLEX. Here, we want to show:

- The effectiveness of our constructive methods;
- The differences between the encodings and strategies;
- The best heuristic strategy for the problem.

Figure 7: No-cross analysis

(a) No crossing analysis with 4000 seconds and cost ratio up to 1.5



(b) No crossing analysis with 4000 seconds and cost ratio up to 25

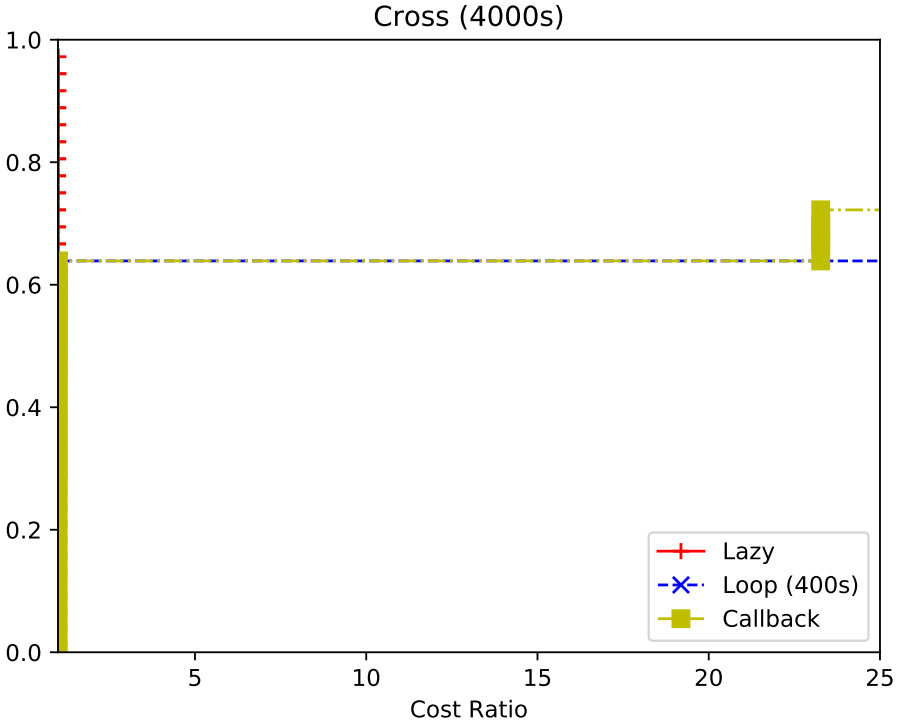


Table 1: No-crossing job: detailed results

Dataset	Lazy	Loop	Callback
data1	19509211.69	20053206.94	19690711.68
data2	21513835.57	21598842.12	21530366.10
data3	22688175.02	22617203.57	22690517.36
data4	24451773.44	24463883.06	24460184.71
data5	23591580.82	23660803.25	23762416.88
data6	25339325.71	26174818.35	1026118558.08
data20	39320932.81	10000000000000.0	1038199951.010
data21	44918761.02	10000000000000.0	1045246871.75
data26	22337935.84	22374602.37	22420709.61
data27	23549225.12	23531634.38	23596892.66
data28	26873708.46	10000000000000.0	1026602385.18
data29	1026803127.37	10000000000000.0	1026917348.38
data1	19511558.28	19466396.26	19690711.68
data2	21514265.04	21714183.48	21530366.10
data3	22688175.02	22687555.68	22690517.36
data4	24451773.44	24463944.97	24470976.55
data5	23591580.82	23580188.43	23762416.88
data6	25347097.82	25231267.79	1026118558.08
data20	39320932.81	10000000000000.0	1038199951.01
data21	44918761.02	10000000000000.0	1045246871.75
data26	22338284.59	22447521.40	22420709.61
data27	23742646.91	23603537.09	23596892.66
data28	26873708.46	10000000000000.0	1026602385.18
data29	1026803128.37	10000000000000.0	1026816485.40
data1	19511558.28	19499256.62	19690711.68
data2	21513835.57	21804163.34	21530124.60
data3	22688175.02	22616832.74	22690517.36
data4	24451773.44	24463845.89	24460184.71
data5	23591580.82	23783590.37	23762416.88
data6	25339325.71	25874725.48	1026118558.08
data20	39320932.81	10000000000000.0	1038199951.01
data21	44918761.02	10000000000000.0	1045246871.75
data26	22338284.59	22338030.88	22420709.61
data27	23742646.91	23519984.24	23596892.66
data28	26873708.46	10000000000000.0	1026602385.18
data29	1026803128.37	10000000000000.0	1026932380.36

4.4.1 Constructive

First of all it is useful to have an understanding about the solutions given in the constructive phase. As an example, figure 8 and 9 show the different techniques described in subsection 2.3.
One thing immediately catches the eye in figure 8: all the flow is given to the substation by only one turbine; clearly, this is a problem (and those two solutions surely can't be provided as final outputs). Instead, we can see that in the optimal solution (fig 10), more than one turbine is linked to the substation. Due to this fact, overall costs (found in the figure title) are very different, due to the added penalties. As it can be seen in figure 8a, this behavior is caused by Prim's algorithm nature and by the wind farm structure: most of the times the turbines are closer to each other than to the substation. This problem is solved using more randomized solutions, that will be fixed later, in the refinement stage. Indeed, why would one randomize the costs? It is possible to find a better solution, because randomization always introduces new possibilities (but the good ones come with small probability), but this is not the case: we need a good amount of sub-optimal solutions to let our refinement phase work properly (as it is explained next). Furthermore, MST algorithms usually find a single solution (tree); even with randomization, the solutions tend to be all the same (although, when the randomization is high, the solutions are simply "chaotic" and make no sense). A mandatory characteristic for the refinement algorithm we have chosen, the genetic algorithm approach, is that the solutions should be "colorful": we would like to give several kinds of solutions as input for the refinement phase.
This justifies the use of our BFS constructive method, which makes very unconvienient choices (see the costs in 9), but such choices are very different from what a MST approach would give. This diversity is what the genetic algorithm we're about to use needs.

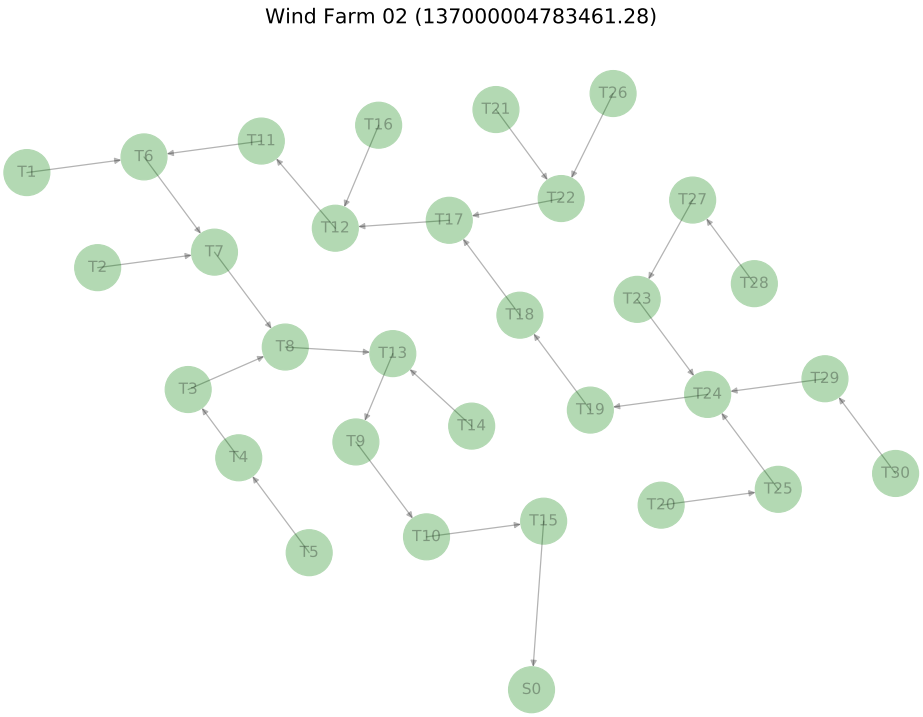
4.4.2 Refinement

As we stated before, our refinement phase purely consists in use of a genetic algorithm. We have set up every parameter for the genetic algorithm, in a way that it should optimize its performance; here, we list the significant settings:

- We've set the population size to be 500, to help the diversification in the breeding phase;
- We let the diversification phase set in when there are 25 unsuccessful iterations/generations (here, "unsuccessful" means to not having improvements either to the alpha male or to the fitness);
- We've set the expected proportion of healthy chromosomes to be very high in the intensification phase (0.995), while it is fair in the diversification phase (0.80);

Figure 8: Examples from constructive algorithms (figures can be zoomed)

(a) The MST



(b) Cost randomization with delta interval set to 0.1

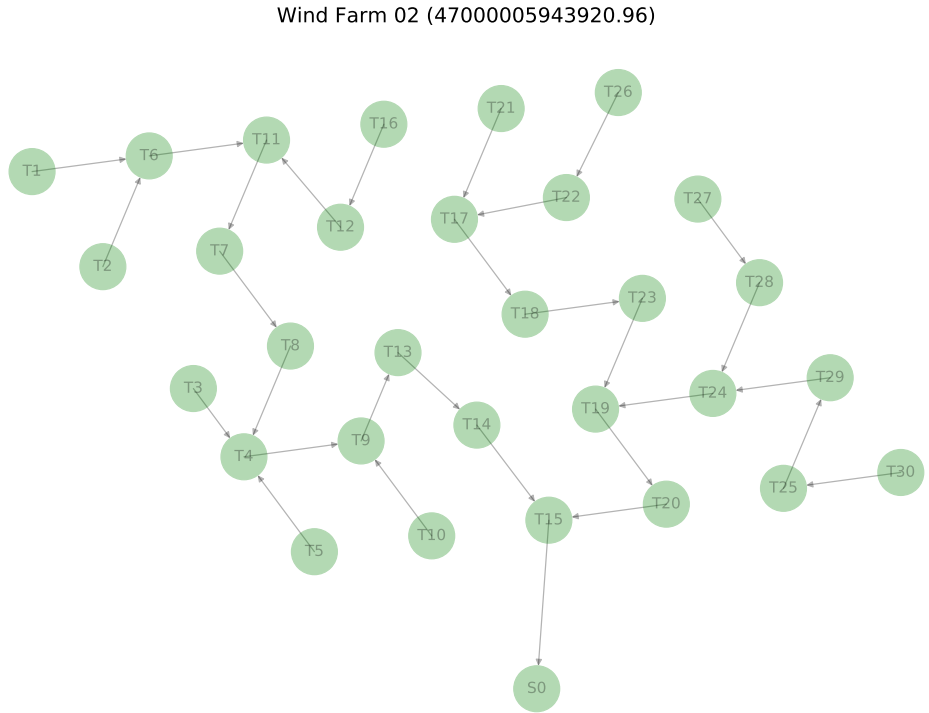
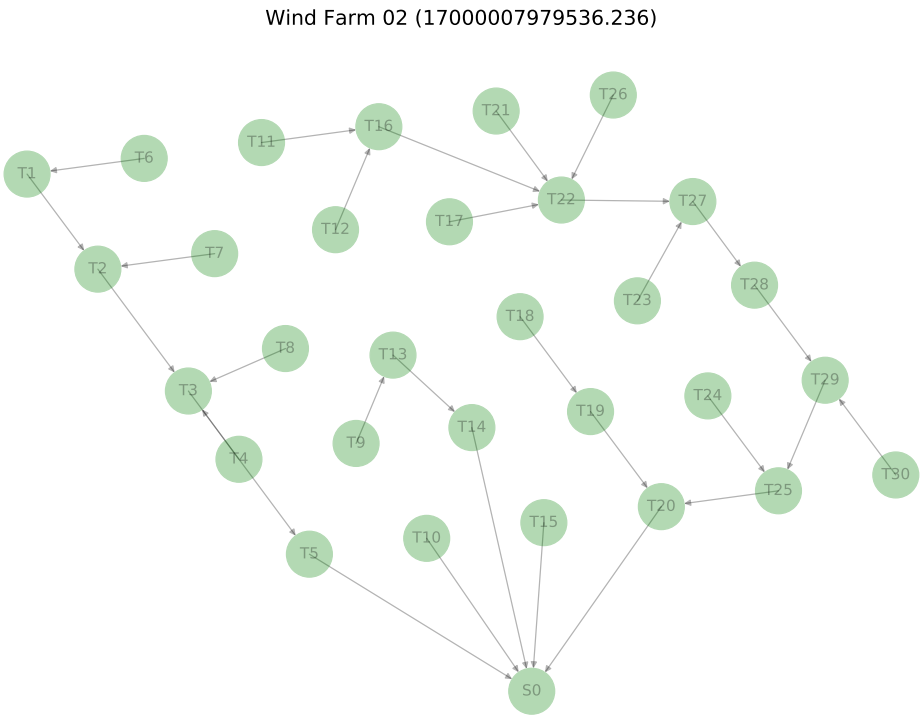


Figure 9: Examples from constructive algorithms (2nd part)

(a) GRASP with $k = 10$



(b) BFS constructive

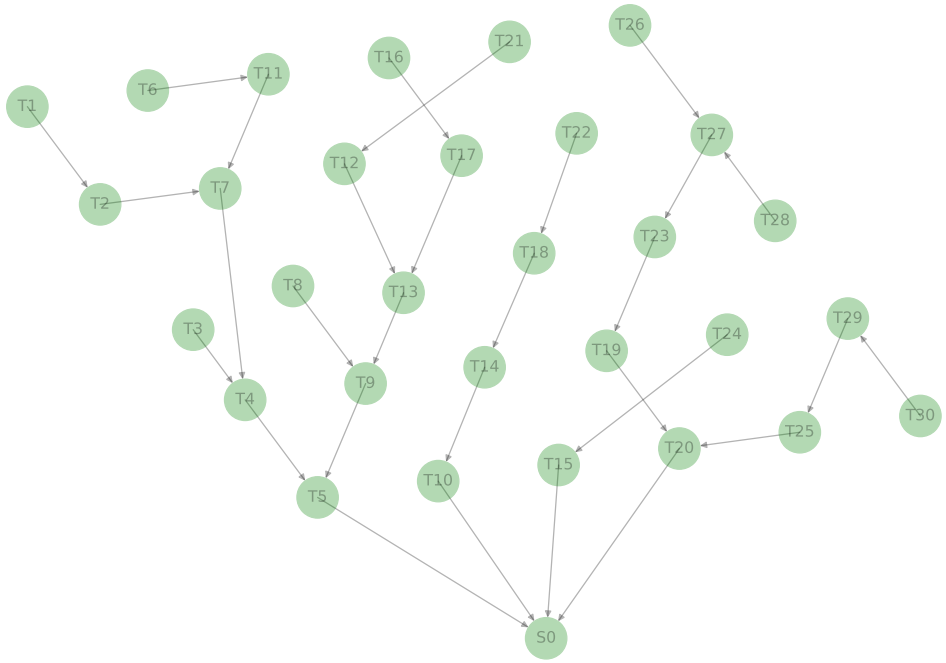
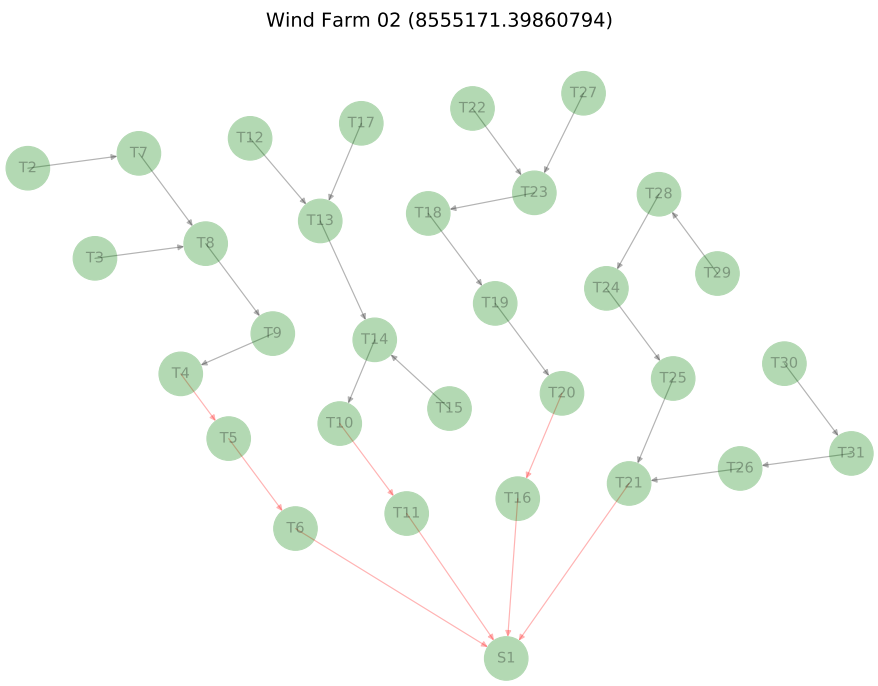


Figure 10: Optimal solution



- We let the mutations occur only when the diversification phase sets in, by selecting the chromosomes to be mutated with probability $\frac{1}{2}$ and single gene mutation probability 0.05.

The previous constructive methods are used at the beginning of the algorithm, as a "start-up phase".
Furthermore, to set one final parameter, we wondered if the constructive methods actually are useful to the genetic algorithm; as we have already explained in subsection 2.4.2, we investigate on the correct choice of the constructive proportion used at the beginning, since we believe it can have an impact. We chose to make a comparison for each encoding chosen (*Prüfer*, *Kruskal* and *Succ*). There is no doubt about the success of the use of the constructive methods, for each encoding, as it is shown in figure 11a, figure 11b and figure 12a.
In figure 12b a plain fact emerges: the *Kruskal* encoding is most likely to out-perform the other two methods.

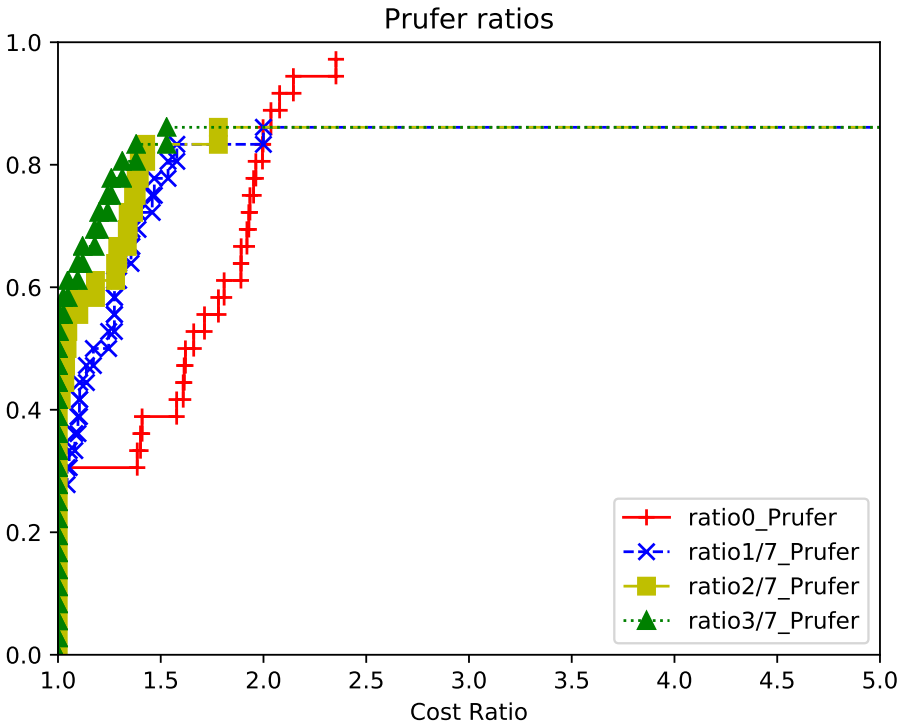
4.5 Comparison between CPLEX and Metaheuristics

Heuristics are often used for three main reasons:

- first, one could not afford CPLEX or an equal reliable MIP solver (these softwares are expensive);

Figure 11: Ratio analysis (part 1)

(a) Ratio analysis on Prüfer's number's based genetic



(b) Ratio analysis on Kruskal's based genetic

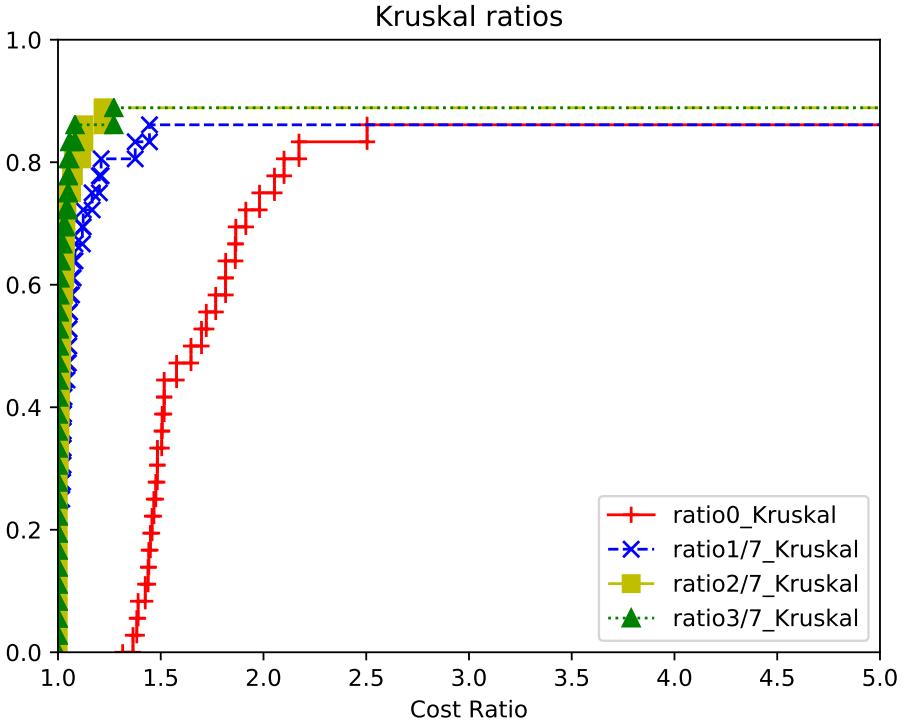
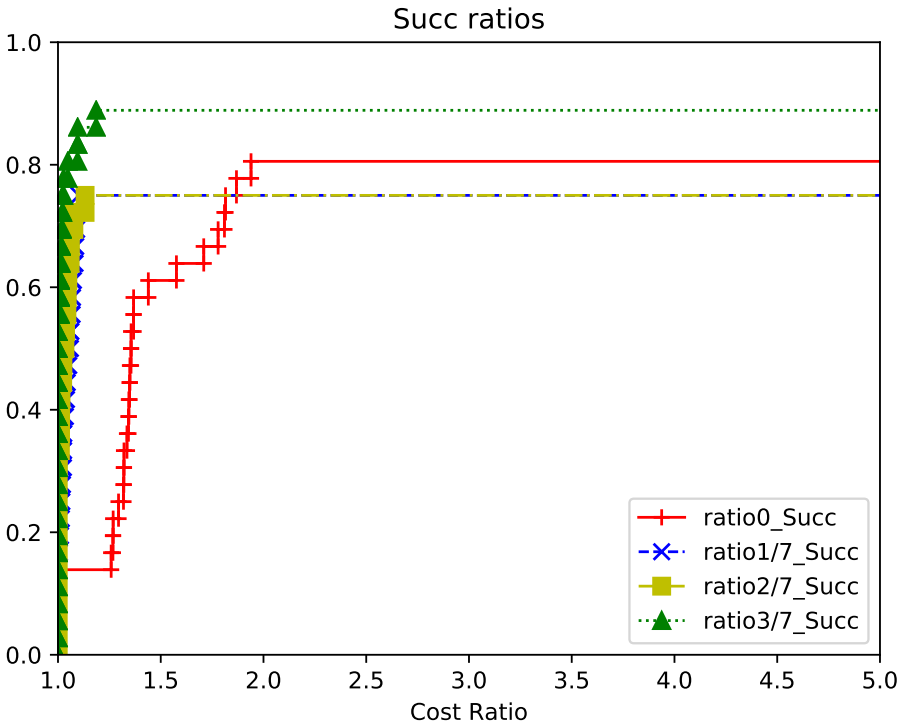


Figure 12: Ratio analysis (part 2)

(a) Ratio analysis on Succ's based genetic



(b) Ratio analysis all combined

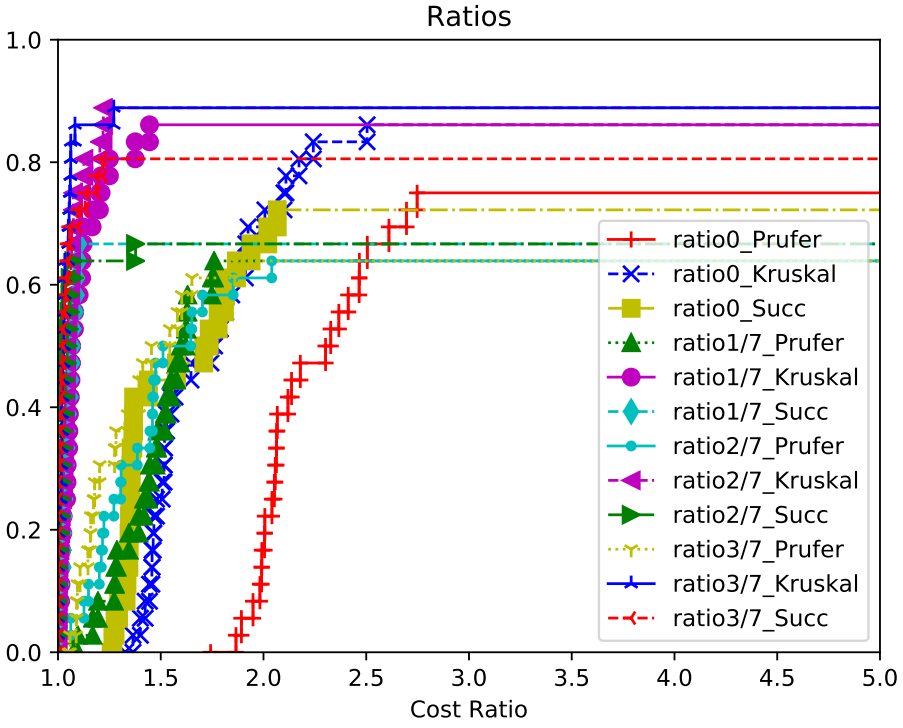
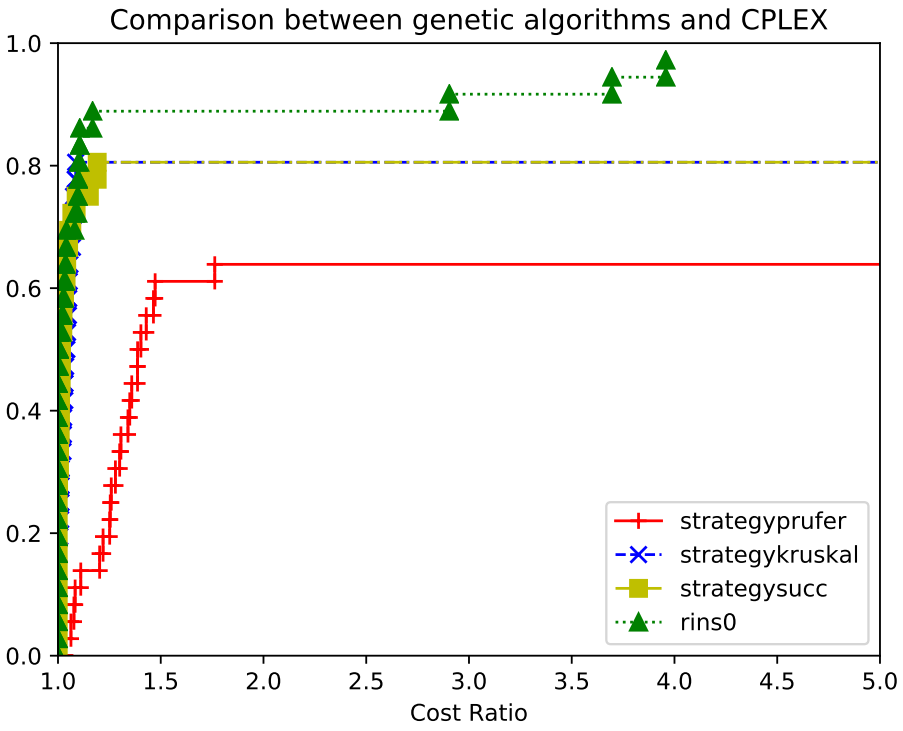


Figure 13: Genetic algorithms and CPLEX

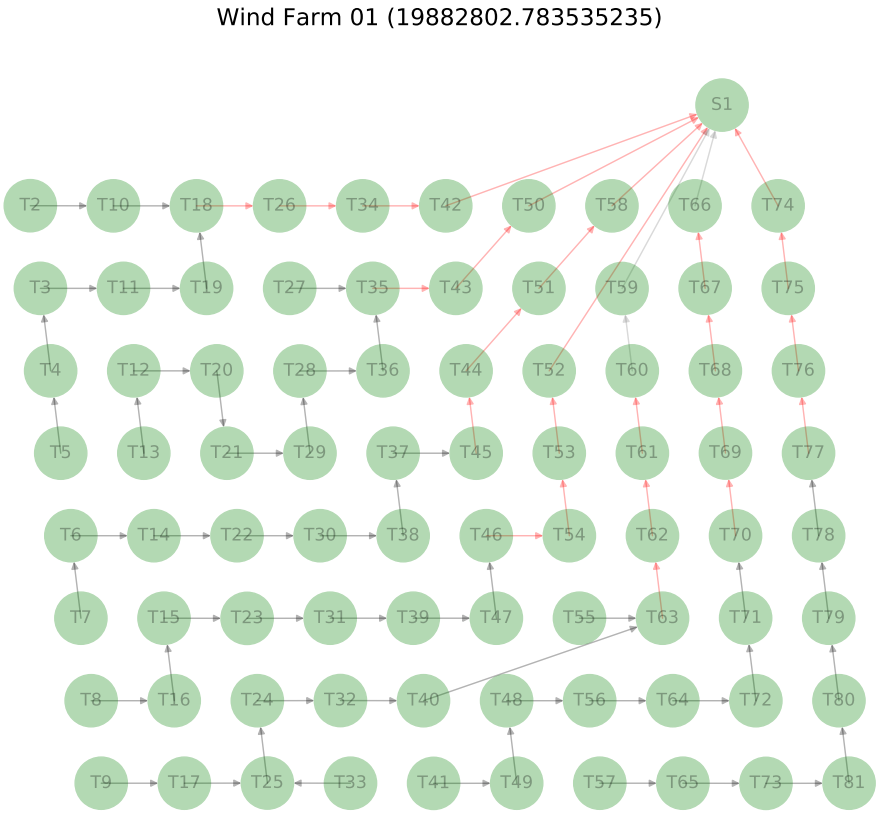


- second, one could not wait the MIP solving times (it takes some reasonable but not short amount of time to let CPLEX find something feasible);
- third, one could set CPLEX with a "warm start" (i.e. giving it some initial solutions, not necessarily feasible);

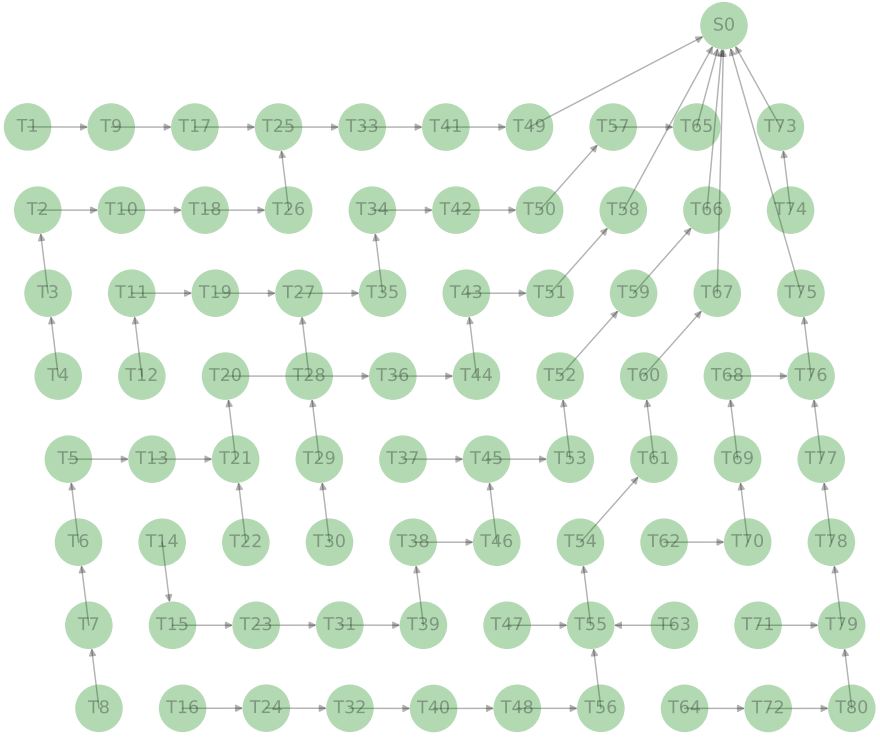
Because of this, it is useful to see whenever it is convenient to use either heuristics or CPLEX.
For a first comparison, during the semester, we have run our genetic algorithm for five minutes on the first instance against CPLEX in the same amount of time. The two given solutions can be found in figure 14a and 14b: the Heuristic solution has a little gap from the CPLEX one.
In figure 13 we show the comparison between CPLEX and the genetic algorithms on every dataset, with the usual performance profiling plot; a timeout has been set to 600 seconds. It is clear that, even if we use its least-efficient configuration, CPLEX has a better overall performance on (almost) every instance. The plot also shows that our heuristics are not performing terribly (besides Prüfer), compared to CPLEX.

Figure 14: Visual comparison on dataset 1 with 300 seconds

(a) Dataset 1 with CPLEX



(b) Dataset 1 with Succ based genetic



5 Conclusions

The course has highlighted and covered many ways and tools to solve the Wind Farm problem. We have chosen to use CPLEX and Python for this aim. Here we describe our experience.

5.1 CPLEX vs Heuristics

First, a short discussion about the performance of the genetic algorithms should be done. There is no doubt that the worst performer of the analysis is the *Prüfer* encoding, which is put in difficulty especially by the all-random-proportion choice. This confirms what has been said by Gottlieb et al. in [11]: Prüfer numbers heavily restrict convergence to a "good" optimal because of the aforementioned problems (see subsection 2.4). Instead, the "winner" is the *Kruskal* encoding, which outperforms the *Succ* one in most of the instances.

When comparing CPLEX to other Heuristics, we've showed that CPLEX has a better overall performance; although we have no doubt that the Heuristics would perform better with less time given, figure 13 shows a strong performance by the MIP solver. This doesn't mean that Heuristics aren't useful; besides the warm-up use of heuristics, their strongest asset is that they can be exploited for solutions in a short window of time.

5.2 Python in OR

Python is gaining momentum in many fields, nowadays it is go the goto language for Machine Learning. It is very useful in both the scientific and industrial world. It is becoming more popular in the OR field too. We chose this language to test its boundaries. Using the Python API we did not find any limitation during the course and we were able to do any request asked. The CPLEX API documentation is very well documented and we had no troubles to understand the different Object Oriented interface. Python offers more protection and usability compared to the C interface. No segmentation fault is raised and this can spare some time during development. The CPLEX engine of course is not written in Python. The Python interface only builds the model and then the computation is done by the standard CPLEX solver.

While the nature of OR (building a model and solving it next) allows to exploit the speed of C and the usability of Python, this does not occur when implementing from scratch a metaheuristic. Python is known for being slow and we have had some troubles when trying to add the no-crossing constraint to the genetic algorithm. The computation was way too long compared to C. Libraries like sci-kit, the well-known library for ML, uses C for intensive tasks, while it uses Python for the user interface. This proves that Python can be combined with C, using softwares like Cython, to provide efficient and usable software.

5.3 Further improvements

This project can be still improved, besides setting CPLEX with a warm start (see 4.5), combining heuristics and CPLEX into one single function; Python, as previously stated, lacks of efficiency, and some good practices can be applied to bypass this problem. Furthermore, we had several ideas that we didn't implement (because we had to choose, since we have time restrictions), but that we think that they are worth analyzing in the future. Actually, we started to implement some of the following ideas.

Efficiency. As described before, we did not try to solve the no-crossing problem with a genetic algorithm, because of efficiency problems. Our code could be sped up by either porting the code to C (by using Cython), or writing again the whole code to C/C++. We have tried to port some critical functions to C; for example, the function *are_crossing* has been ported, which, given the edges, returns true if they're crossing. This function is called a lot of times during the computation and must be efficient. Our attempt with Cython allowed us to speed up the code up to $x2.5$ times, which is still not sufficient. A good work would need to write again the whole algorithm, or at least the critical functions. Another thing that should be done for the sake of efficiency and for the crossing problem is a $O(n^4)$ sized lookup-table, containing, for each edge pair, a "True" value if they're crossing.

Over-fitting. An error we've made is to focus too much on the first instance (i.e. *dataset_1*), when we needed to properly set some parameters. Indeed, during class, we were induced to think that that was the hardest instance for CPLEX. Even if it was the case, our heuristics (and therefore their parameters) shouldn't be oriented to a single dataset. This is probably the reason why the more customized methods (e.g. the custom repair procedure implemented with the *Succ* encoding) have a better performance on instance number one, while it performs worse on the other datasets.

Genetic algorithm parameters. A genetic algorithm has several parameters. Having more parameters can introduce some flexibility for our algorithm, because, if they're correctly set, they greatly improve the success and the efficacy of the method. However, having many parameters intrinsically introduces the problem of setting them correctly. One idea we've had is to simply give up on some parameters, by finding a parameter-free procedure for each phase of the genetic algorithm. But because these parameters are game-changing when it comes to performance, this simply shouldn't be done. Therefore, since each parameterization optimizes the performance for each dataset, another idea we've had would be to borrow the *Machine Learning* approach (in which the parameters are set automatically, with certain procedures not discussed in this context). This may sound fancy, but it would be onerous (and probably ineffective) to set this quantity of parameters on such small dataset, with such a slow-evaluating objective function.

Genetic algorithm intensification and diversification phases. The way our two distinct phases work are explained in subsection 2.4, but more ideas came along while writing down the code. One first idea was to better distinguish between the two phases: while the intensification phase seem to work well, it looks the diversification one has room for some improvements. To do so, when it looks like that the algorithm is stuck in a local minimum, we would have stopped the breeding-selection step and invoked another function for some generations (i.e. iterations). We would have introduced an external population, primarily made of random trees, to the current one, as an analogy with the *immigration* phenomena. Even here, there are several ways to do so. As a first approximation, we thought about introducing a very small random population and make everyone of the old population mate with the new one. Then, as always, the selection and the mutation phases would set in.

Genetic algorithm intrinsic mechanics. One last change on the genetic algorithm would be to modify the breeding mechanics. First, a small population should be kept, in order to let this change work; something like 30 to 50 specimen would be more than enough. Then, the breeding phase would let every single chromosome to mate with every one. This causes to have $\frac{n \cdot (n-1)}{2} = O(n^2)$ children for each generation, which slows down the computation a lot if n is too large. There is no guarantee that the intensification phase would widen (and therefore find better solutions), since, while the number of children greatly increases, the number of the actual population maintained is very low.

Wind Farm's Hull. One last and very imaginative idea we've had was to build another problem-oriented heuristic method, to help the diversification of the starting population fed to the genetic algorithm. Since the whole energy should flow from the turbines to the substation, our idea was to identify the geometric convex hull of the wind farm. By this we mean that we'd identify the outermost nodes of the farm, representing (somehow) the "boundaries" of the geometrical shape the wind farm has. This can be easily done with several efficient algorithms (e.g. *Quick Hull*, a well known D&C algorithm). Similar of what we do on the BFS constructive approach, we would indeed start from the turbines that are part to the hull: such turbines would make the starting set, from which we'd start to link, with some randomness, nearby turbines. This method has been judged hard and long to implement, with no guarantees to outperform BFS, GRASP or the randomized MST.

Appendices

A Guidelines to use CPLEX with Python 3

Some basic configuration is needed to use CPLEX with Python.

A.1 On local computer

For the sake of clarity we provide full instructions here. This setting has been tested on Ubuntu 16.04 with Python 3.5.2. Later on XXX, will stand for the version of CPLEX downloaded (e.g. version 12.8 => XXX=128). References can be found at [2] [1].

1. Download the bin file from the cplex marketplace.
2. From the terminal, in the folder where the .bin file is stored launch:
   `chmod +x cplex_studioXXX.linux-x86-64.bin`
3. Then type: `sudo ./cplex_studioXXX.linux-x86-64.bin`. During the installation leave everything as default.
4. Change the folder, where CPLEX has been installed.
   `cd /opt/ibm/ILOG/CPLEX_StudioXXX/python`
5. Then launch: `sudo python3 setup.py install`
6. Open the file .bashrc in the home folder and add:

```
export
 ↪  PATH="$PATH:/opt/ibm/ILOG/CPLEX_StudioXXX/cplex/bin/x86-64_linux"
export PATH="$PATH:/opt/ibm/ILOG/CPLEX_StudioXXX/cpoptimizer/bin/x86-6⌋
 ↪  4_linux"
```

The setup can be tested:

1. Open the terminal and type: `python3`
2. In the Python console: `import cplex`

A.2 On 'Blade' Cluster

The department gives the possibility to launch Python applications with CPLEX as well. The latest version installed version on the cluster is the 12.6.3. Since there are several installations of CPLEX a few parameters must be set in order to launch a Python application. The best way is to use the *PYTHONPATH* environment variable. Setting the PYTHONPATH

environment variable is an easy way to make Python modules available for import from any directory. This environment variable can list one or more directory paths which contain your own modules.

To launch a script on one runner on the cluster a file *commands.job* must be created. The first line should be:

```
export PYTHONPATH=$PYTHONPATH:/nfsd/opt/CPLEX12.6/cplex/python/3.4/x86-64_l
↪ inux/
```

In this way Python will search for the CPLEX' libraries inside the folder. For example, here is reported our *commands.job* file:

```
export PYTHONPATH=$PYTHONPATH:/nfsd/opt/CPLEX12.6/cplex/python/3.4/x86-64_l
↪ inux/
python3 src/__init__.py
```

B Cluster script

We have found very useful this script to update our remote repository and launch jobs on the cluster. Here it is sketched with the main functions, but it can be easily extended with ad-hoc features. Here exceptions are not handled, full code is in our repository! Some useful remarks:

- Use Python3 to launch the script
- The repository must be already cloned on the cluster
- Iterate over the desired options to be tested adding loops at line 48
- The exec_command function must include in the same call the export of the SGE_ROOT environment variable and the qsub command
- The -cwd option (current working directory) must be put before the name file of the instruction file

```
1 #!/usr/bin/env python3
2 import os
3 import time
4 import datetime
5 import paramiko
6
7 # All the options for the job
8 dataset_numbers = [1, 2, 3, 4, 5, 6, 20, 21, 26, 27, 28, 29]
```

```
interfaces = ['cplex']
rins_options = [-1, 0, 10, 100]
num_iterations = 3

server = "login.dei.unipd.it"
# Creating a new ssh instance
ssh = paramiko.SSHClient()
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())

# Ask the user the password
username = str(input("Please, enter your username (@" + server + "): "))
pwd = str(getpass.getpass(prompt="Password: "))

# Connect to server
ssh.connect(server, username=username, password=pwd)

# Open secure file transfer protocol instance
sftp = ssh.open_sftp()

# Remote project path
remote_path = "/home/" + username + "/OR2/"

# Get parent directory name
local_path = os.path.dirname(os.getcwd()) + "/"

# Create custom folder
current_time = time.time()
timestamp = datetime.datetime.fromtimestamp(current_time).strftime('%Y-%m-%
↪  d_%H:%M:%S')
current_folder = "run__" + timestamp
sftp.mkdir(remote_path + "out/" + current_folder)

# Files to be uploaded
files = ['src/__init__.py', 'src/lib/WindFarm.py', 'src/lib/callback.py']

# Create a local file that will be sent to the server
with open("commands.job", "w") as fp:
    fp.write("#!/bin/bash \n")
```

```
46      fp.write("export PYTHONPATH=$PYTHONPATH:/nfsd/opt/CPLEX12.6/cplex/pytho
        ↪ n/3.4/x86-64_linux/\n")
47
48      for it in range(num_iterations):
49          for d in dataset_numbers:
50              for i in interfaces:
51                  # Formatting/constructing the instruction to be given:
52                  instruction = "time python3 " + remote_path +
                    ↪ "src/__init__.py"
53                  # Options to be added:
54                  instruction += " --dataset " + str(d)
55                  instruction += " --interface " + i
56                  instruction += " --rins " + str(r)
57                  instruction += " --timeout 400"
58                  instruction += '\n'
59                  fp.write(instruction)
60
61  print("Updating files")
62  for file in files:
63      file_remote = remote_path + file
64      file_local = local_path + file
65
66      print(file_local + ' >>> ' + file_remote)
67      try:
68          sftp.remove(file_remote)
69      except IOError:
70          print("File was not on the cluster")
71      sftp.put(file_local, file_remote)
72
73  # Put the file on the current folder on the cluster and delete the local
    ↪ one
74  print(local_path + 'scripts/commands.job' ' >>> ' + remote_path + 'out/' +
    ↪ current_folder + '/commands.job')
75  sftp.put(local_path + 'scripts/commands.job', remote_path + 'out/' +
    ↪ current_folder + '/commands.job')
76  os.remove("commands.job")
77
78  # Give this job to the cluster
```

```
79 ssh_stdin, ssh_stdout, ssh_stderr = ssh.exec_command("export
   ↪ SGE_ROOT=/usr/share/gridengine \n" + "cd {0}out/{1}
   ↪ \n".format(remote_path, current_folder) + "qsub -cwd commands.job")
80
81 # Print output and errors
82 print(ssh_stdout.read().decode('utf-8'))
83 print(ssh_stderr.read().decode('utf-8'))
84
85 sftp.close()
86 ssh.close()
```

References

[1] Installation of ibm ilog cplex optimization studio on linux and mac os platforms. URL http://www-01.ibm.com/support/docview.wss?uid=swg21444285.

[2] Setting up the python api of cplex. URL https://www.ibm.com/support/knowledgecenter/SSSA5P_12.8.0/ilog.odms.cplex.help/CPLEX/GettingStarted/topics/set_up/Python_setup.html.

[3] Marco A. Boschetti, Vittorio Maniezzo, Matteo Roffilli, and Antonio Bolufé Röhler. Matheuristics: Optimization, simulation and control. In María J. Blesa, Christian Blum, Luca Di Gaspero, Andrea Roli, Michael Sampels, and Andrea Schaerf, editors, *Hybrid Metaheuristics*, pages 171–177, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-04918-7.

[4] Ilhem Boussaïd, Julien Lepagnot, and Patrick Siarry. A survey on optimization metaheuristics. *Information Sciences*, 237:82 – 117, 2013. ISSN 0020-0255. doi: https://doi.org/10.1016/j.ins.2013.02.041. URL http://www.sciencedirect.com/science/article/pii/S0020025513001588. Prediction, Control and Diagnosis using Advanced Neural Computations.

[5] Emilie Danna, Edward Rothberg, and Claude Le Pape. Exploring relaxation induced neighborhoods to improve milp solutions. 102:71–, 01 2005.

[6] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002.

[7] Thomas A. Feo and Mauricio G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6(2):109–133, Mar 1995. ISSN 1573-2916. doi: 10.1007/BF01096763. URL https://doi.org/10.1007/BF01096763.

[8] Martina Fischetti and David Pisinger. Optimizing wind farm cable routing considering power losses. *European Journal of Operational Research*, 2017. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2017.07.061. URL http://www.sciencedirect.com/science/article/pii/S037722171730704X.

[9] Matteo Fischetti. *Lezioni di Ricerca Operativa*. 2014.

[10] Matteo Fischetti and Andrea Lodi. Local branching. *Mathematical Programming*, 98 (1):23–47, Sep 2003. ISSN 1436-4646. doi: 10.1007/s10107-003-0395-5. URL https://doi.org/10.1007/s10107-003-0395-5.

[11] Jens Gottlieb, Bryant A. Julstrom, Günther R. Raidl, and Franz Rothlauf. Prüfer numbers: A poor representation of spanning trees for evolutionary search. In

Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation, GECCO'01, pages 343–350, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1-55860-774-9. URL `http://dl.acm.org/citation.cfm?id=2955239.2955297`.

[12] *IBM ILOG CPLEX Optimization Studio. Getting started with CPLEX*. IBM.