Università Degli Studi Di Padova
Course in Operations Research 2

Wind Farm Cables Routing Optimization

Massimo Meneghello and Andrea Tommasi


March 8, 2019

Contents

1 Introduction

Wind energy is a source of renewable power that comes from air currents flowing across the earth's surface. Nowadays it is one of the new fastest growing sources of electricity in the world. As the number of turbines per wind farm increases, it becomes extremely important to optimize the cable connections among them: on one hand, to lower installation costs, on the other, to reduce energy losses during operation.

In this paper, we first present a Mixed Integer Linear Programming (MILP) model that is able to handle most of the required real-world constraints. Subsequently, we show several optimization approaches to solve this problem, and we report the collected data on different existing wind farms.

For a more exhaustive account on wind farm optimization, see [6]: we used this article as a benchmark for our work, developing the same mathematical model and using the same test bed in order to compare the results.

In chapter 2, we present a comprehensive description of the mathematical model we used, providing details regarding peculiar constraints and relaxed versions of the model.

Chapter 3 focuses on solving the MILP model. In the first part, we present a brief introduction to the `CPLEX` optimization software and we show how to use its main features, in particular, the *lazy constraints* and the *callbacks*. Then, we provide a solution that uses `CPLEX` as a *black box*, called the *Loop Method*.

In chapter 4, we show a matheuristic strategy, the *Hard-fixing* and two heuristic approaches we entirely designed and coded. In this case, we decided to use a *multi-start* (MS) approach and the *Variable Neighborhood Search* (VNS) heuristics.

Finally, chapter 5 contains information about the results we collected in our experiments and the main differences with respect to the results Fischetti and Pisinger reported in their work [6].

2 MILP Model

In each offshore wind farm, one or more substations collect all the power of the turbines and convey this energy to the mainland. Each turbine must be joined through a cable to another turbine or to a substation. The final layout can be represented with a *tree* structure (or a *forest*, in presence of many substations), where all the nodes are connected to one substation (the root of the tree). This layout is called *cables routing*.

In this article, we do not take into account the problem of positioning the turbines. In fact, we only analyze real instances in which the turbines have already been placed. A specific description of this subject is instead provided in [5].

Additionally, in order to simplify the representation of the problem, all examples we consider in this report involve only one substation.

Therefore, assuming that the best turbine positions have been identified, we aim at finding an optimal cables routing among all the turbines and the substation, minimizing the total cables amount.

2.1 Basic Model

In the mathematical model in [6], the following constraints are taken into account.

The first constraint to satisfy is the balance of flows: the energy flow entering a turbine must be equivalent to the outgoing one, plus the turbine's production. Moreover, only one cable can come out of each turbine, so we must have a flow at most equal to the capacity of the biggest cable provided because every cable installed must withstand the power coming out of the turbine.

Another limitation is the substation's capability, named the maximum number of cables that the substation can accept. In our testbed, the substation has a parameter, indicated as C, representing this value.

All the instances we consider are composed of a wind farm layout and a set of cables that can be used for the layout: each cable presents a different cost and capacity. In the following, we say n to be the number of turbines (including the substation) in the layout and we indicate as K the cardinality of the cables' set.

We can describe the cables routing as a direct graph $G = (V, A)$, with $V = \{1, \dots, n\}$ representing the set of turbines and A denoting the set of arcs. We recall that in a direct graph $(i, j) \neq (j, i)$, while $\{i, j\}$ stands for (i, j) and (j, i).

Every arc must cope a *flow* that represents the energy captured by the turbines in the subtree from which the arc is coming out. We represent the flow running from i to j with the variable

$$f_{i,j} \geq 0, \ \forall (i, j) \in A. \tag{1}$$

The second class of variables in our model indicates whether an arc $(i, j) \in A$

is build using a cable of type k:

$$x_{i,j}^k = \begin{cases} 1 & \text{if arc } (i,j) \in A \text{ is built with a cable of type } k \\ 0 & \text{otherwise.} \end{cases} \quad \forall\, (i,j) \in A, \forall\, k \,:\, 1 \leq k \leq K \tag{2}$$

Our last type of variables indicates whether an arc (i,j) of G is selected for the cables routing or not:

$$y_{i,j} = \begin{cases} 1 & \text{if arc } (i,j) \in A \text{ is selected for the cables routing} \\ 0 & \text{otherwise.} \end{cases} \quad \forall\, (i,j) \in A \tag{3}$$

Regarding the model's constraints, the first type of constraints is the one imposing that an arc (i,j) can be built using just one type of cables.

$$y_{i,j} = \sum_{k=1}^{K} x_{i,j}^k, \quad \forall\, (i,j) \in A. \tag{4}$$

Next, we add out-degree constraints for turbines and substation: we desire that only one cable comes out from each turbine. The P_h value indicates the energy produced by the turbine h: if this value is equal to -1 means that h is the substation and no cable must exit from it.

$$\sum_{j=1}^{n} y_{h,j} = \begin{cases} 1 & \text{if } P_h > 0 \\ 0 & \text{if } P_h = -1 \quad \text{(substation)} \end{cases} \quad \forall\, h \in V \tag{5}$$

With the subsequent, we want to pose a restriction to the number of cables entering the substation and we want this quantity to be less than or equal to parameter C:

$$\sum_{i=1}^{n} y_{i,h} \leq C, \quad \forall\, h \in V : P_h = -1. \tag{6}$$

In order to avoid self-loops, we also impose that

$$y_{i,i} = 0, \quad \forall\, i \in V \tag{7}$$

Another significant rule to catch is the balance of flows: the flow entering a turbine must equal the outgoing one plus the power P_h produced by the turbine h.

$$\sum_{j=1}^{n} f_{h,j} = \sum_{i=1}^{n} f_{i,h} + P_h, \quad \forall\, h \in V : P_h \geq 0. \tag{8}$$

The energy flow leaving a turbine must be supported by a single cable, consequently, the capacity of the cable from turbine i to j must be greater or equal to the flow $f_{i,j}$ coming out from turbine i. Saying $cap(k)$ be the maximum capacity of type k cable, this constraint can be expressed as:

$$\sum_{k=1}^{K} cap(k) \cdot x_{i,j}^k \geq f_{i,j}, \quad \forall\, (i,j) \in A. \tag{9}$$

Finally, what we desire to minimize is the total amount of posed cables, considering as well the cost of the different types of them. Saying $dist(i,j)$ be the Euclidean distance between turbines i and j and $cost(k)$ be the cost of type k cable for length unit, the objective function we obtain is

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{K} cost(k)\cdot dist(i,j)\cdot x_{i,j}^{k} \tag{10}$$

The entire MILP model is summarized below:

$$\min \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{K} cost(k)\cdot dist(i,j)\cdot x_{i,j}^{k} \tag{11}$$

$$\sum_{k=1}^{K} x_{i,j}^{k} = y_{i,j}, \quad \forall\,(i,j)\in A \tag{12}$$

$$\sum_{l=1}^{n}(f_{h,l}-f_{l,h}) = P_h, \quad \forall\, h\in V : P_h \geq 0 \tag{13}$$

$$\sum_{k=1}^{K} cap(k)\cdot x_{i,j}^{k} \geq f_{i,j}, \quad \forall\,(i,j)\in A \tag{14}$$

$$\sum_{j=1}^{n} y_{h,j} = 1, \quad \forall\, h\in V : P_h \neq -1 \tag{15}$$

$$\sum_{i=1}^{n} y_{i,h} \leq C, \quad \forall\, h\in V : P_h = -1 \tag{16}$$

$$y_{i,i} = 0, \quad \forall\, i\in V \tag{17}$$

$$x_{i,j}^{k}\in\{0,1\},\ \forall\,(i,j)\in A,\ \forall k,\ 1\leq k\leq K \tag{18}$$

$$y_{i,j}\in\{0,1\},\ \forall\,(i,j)\in A \tag{19}$$

$$f_{i,j}\geq 0,\ \forall\,(i,j)\in A \tag{20}$$

2.2 No-Cross Constraints

The model we just defined lacks of a truly relevant point, in fact, a potential solution could present some cables that intersect each other. During the cables' laying, this event could involve several complications, for example, supplementary costs should be added, or there could be difficulties in making bridges that could also lead to the ruin of the cables themselves.

This means that we have to optimize the problem avoiding intersections in the final layout. Therefore, a new class of constraints is necessary to avoid this unwanted event but, given the particular nature of these constraints and their high number, these are treated separately.

At first glance, the number of no-cross constraints appears to be in order $O(n^4)$: indeed, it seems necessary to check if each edge in A, which number is $O(n^2)$, intersect other edges.

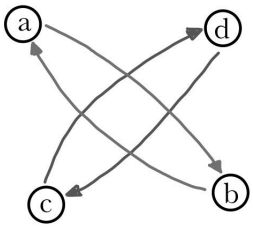

Figure 1: (no-cross constraints explained) if there is a cable connecting a and b, a cable starting (or ending) in c cannot be laid if this one passes between a and b.

However, following [6], the number of constraints can be reduced to be $O(n^3)$ considering triples of points. Let (a, b, c) be such a triple, then, if a cable has been set between a and b, it is necessary to avoid introducing cables starting (or ending) in c and crossing the edge $\{a, b\}$.

A more exhaustive description of these constraints and a formal proof of their validity for the model can be found here [6]. Adhering to this description, let $\mathcal{C} \subset A \times A$ be *the set of couples of edges that intersect each other*. We can define the clique arc subset $\mathcal{Q}$ as

$$\mathcal{Q}(a, b, k) = \{(a, b), (b, a)\} \cup \{(k, h) \in A : (\{a, b\}, \{k, h\}) \in \mathcal{C}\} \tag{21}$$

Finally, we introduce the no-cross constraints:

$$\sum_{(i,j)\in\mathcal{Q}(a,b,k)} y_{i,j} \leq 1 \quad a, b, k \in V, \, |\{a, b, k\}| = 3 \tag{22}$$

A note about performance. Although with the method we described the amount of constraints is reduced to $O(n^3)$, the number of operations required to compute them is still $O(n^4)$.

Clearly, this is irrelevant if the constraints are generated once. However, some applications we developed need to generate or check no-cross constraints numerous times. In this case, we think the best solution is to implement a *look-up table* that, for every couple of edges, returns if they intersect or not.

Let consider the instance with $n = 100$ turbines (worst case in this report), in order to save the information for the table it would be necessary to maintain 100^4 *bytes* (approximately 95 $Mbytes$). By wasting one byte to store a boolean value, we do not achieve satisfying management of space. By means of more efficient data structures, we reduced that quantity to approximately 3 $Mbytes$.

Although this could seem quite a large value, it is sufficient to fit the look-up table in the cache of most commercial processors.

2.3 Relaxed Model

We introduce now a relaxed variant of the model previously described: this allows `CPLEX` to find an initial incorrect solution, namely a solution violating

the model (11) - (22), yet in a short amount of time. Nevertheless, the cost of a relaxed solution will be such that `CPLEX` will always look for a solution restricted by constraints of the original model.

In order to obtain such variant, we include two kinds of relaxed constraints:

- *substation in-degree relaxation*, that allows the substation to receive more input cables, violating the limit imposed by C;
- *cable flows relaxation*, with this artifice it is possible to admit disconnected solutionts.

For the first one, we change equation (6) introducing the slack variable s

$$\sum_{i=1}^{n} y_{i,h} - s \leq C, \quad \forall\, h \in V : P_h = -1 \tag{23}$$

$$s \geq 0,\ s\ integer. \tag{24}$$

For the second relaxation, we modify equations (8) as:

$$\sum_{l=1}^{n} (f_{h,l} - f_{l,h}) + l_h = P_h, \quad \forall\, h \in V : P_h \geq 0 \tag{25}$$

$$0 \leq l_h \leq k_{MAX}, \quad \forall\, h \in V : P_h \geq 0 \tag{26}$$

where k_{MAX} is the maximum cable capacity.

The coefficients of variables s and l_h in the objective function are fixed to $M \gg 0$, a sufficiently large value.

3 Implementing and Solving the Model with CPLEX

In order to solve the problem introduced earlier, we used different types of algorithms. As can be seen from section 2.2, the main obstacle for a solver is represented by the high number of constraints used to deny that the cables intersect each other. Even for a state-of-the-art LP and MILP solver as `CPLEX`, it could be surprisingly difficult to obtain an acceptable solution in a reasonable amount of time when the number of turbines is greater than 70 or 80.

Consequently, to circumvent this complication, we developed and analyzed different approaches. Some of these methods require `CPLEX`, however, we also provided heuristic algorithms.

The algorithms can be divided into three classes, each representing a different paradigm:

- *exact*;
- *matheuristic*;
- *heuristic*.

Solvers handling exclusively `CPLEX` features belong to the first class. These ones try to solve the problem optimally, but they could employ a huge amount of time to prove that a solution is the optimal one.

Instead, algorithms using `CPLEX` in order to obtain intermediate solutions, reside to the matheuristic class. With this approach, optimality is not guaranteed still it is much simpler to find a valid solution.

At last, we dedicate the next chapter to explain and analyze our heuristic algorithms.

3.1 Extact Solvers

`IBM ILOG CPLEX` [10] is an extremely powerful optimization software providing a large number of features and configurations. `CPLEX` can be used as an interactive solver, indeed thanks to its command line interface, the user can furnish a file containing a model, set parameters and ask `CPLEX` to find a solution. However, `CPLEX` provides lower level accesses interfacing with most widespread programming languages, such as C, C++, Java, C# and Python.

In the present work, we exclusively discuss the approach with the *CPLEX Callable Library* (CCL) that is the interface provided for the C and C++ programming languages.

Basic Guide to CPLEX

As can be guessed from the title, this does not want to be an exhaustive guide to all the secrets concerning `CPLEX` but rather a tutorial that helps a novice user in modelling problems with this software. For a more detailed discussion, we recommend consulting the *CPLEX User's Manual* [10] and the product website [2].

First of all, it is necessary to define an environment in which to operate and the `CPXopenCPLEX` function fulfils this task. The type `CPXENVptr` we obtain represents an environment that keeps information about the parameters we set.

The next step requires the initialization of a new problem with `CPXcreateprob` that returns a `CPXLPptr` pointer: this represents the key to access all data regarding our problem throughout the code we develop. These two components we just created (or at least the `CPLEX` environment) are required by every `CPLEX` function.

Next, we need to generate the MILP model or, in other words, we require a procedure to translate the abstractions of the mathematical model into constraints and variables that `CPLEX` is able to deal with. Conventionally, in linear programming a variable is represented by a column of the matrix while a constraint is represented by a row. Adopting this convention, the CCL provides the `CPXnewcols` function to add one or more columns to the model.

This function allows the user to set:

- the **number of columns** the user wants to add;
- the **objective function coefficients**;
- the **lower bounds**;
- the **upper bounds**;
- the **types** of the variables (*binary*, *continuous*, *integer*, *semi-continuous* or *semi-integer*);
- the **names**, this makes it easier to distinguish a variable.

According to model (11) - (26), we divide variables into five classes, assigning them different names:

- $f(i, j)$, continuous non-negative variables representing the flow running in the edge $(i, j) \in A$;
- $x(i, j, k)$, binary variables representing the choice of the cable in the edge $(i, j) \in A$, $\forall\, k$, $1 \leq k \leq K$;
- $y(i, j)$, binary variables representing if the edge $(i, j) \in A$ must be built or not;
- s, integer non-negative variable, it is used to relax the substation input degree constraint;
- $l(h)$, continuous non-negative variables $\forall\, h \in V$, these are employed to relax the flow constraints.

The coefficients of the f and y variables are clearly 0, instead, the coefficients of the $x(i, j, k)$ variables are given by the length of cable (i, j) multiplied by the unitary cost of the k-th cable. As said in their dedicated subsection, the s and l variables should not be chosen by the solver, thus their coefficient must be a very high value and we retain 10^{12} could be large enough for our purposes.

The total number of variables is $n^2(K + 2) + n + 1$ and this quantity is definitely dominated by the number of the x variables.

Within CPLEX a constraint is represented by a row, then the CPXnewrows function permits the user to add one or more constraints. It is possible to set the following parameters:

- the **number of rows** the user wants to add;
- the **right-hand-sides** of the rows;
- the **senses** (*lower or equal*, *equal*, *greater or equal*, *ranged*);
- the **range values**;
- the **names**.

When a new row is appended to the model, its coefficients are fixed to zero by default. The CPXchgcoef routine is required to change one coefficient of the matrix providing the **row's index**, the **column's index**, and the coefficient's **new value**. Other useful functions in this phase are CPXgetnumcols and CPXgetnumrows that respectively provide the current number of columns and the current number of rows constituting the matrix.

Once understanding the theoretical model, the implementation of its constraints through CPLEX should be quite straightforward, therefore we do not spend further words in technical details.

We consider the default model for our purposes the one composed of the (11) - (20) equations while the (23) - (26) equations can be added or removed by fixing their two related parameters in our software. No-cross constraints (22) are not included in the default model because we developed several procedures to append them and each procedure possesses pro and cons we desire to analyse.

Before trying to launch the solver, the user can choose to set several among the large number of parameters CPLEX makes available. This can be done by using CPXsetintparam, CPXsetlongparam or CPXsetdblparam functions depending on whether the parameter we want to access is respectively an integer, a long integer or a double type.

For example, one of the most common parameters a user would like to fix is certainly the computation time limit. The command

```
CPXsetdblparam(env, CPX_PARAM_TILIM, 60.0);
```

sets parameter CPX_PARAM_TILIM (that is a double type) to 60 seconds in the env environment.

The calculation starts when the CPXmipopt function is called. At this point, other choices are available, for example, if a model does not contain binary or integer variables, the CPXlpopt routine must be used.

At the expiry of the time limit chosen by the user, CPLEX may not have found a solution, in this case, the CPXgetx function throws an error. Otherwise, it allows getting a vector containing the values of each variable in the best solution found until that time. Other useful information at the end of the computation can be found with CPXgetbestobjval, that returns the value of the best known bound on the optimal solution value, and CPXgetobjval, that provides the solution objective value.

The user can also save the model to an *LP file*[1] in every moment after generating it with the `CPXwriteprob` routine.

Finally, when the model is no longer needed, the user must free it with `CPXfreeprob`. Afterwards, it is necessary to call the `CPXcloseCPLEX` function to close the `CPLEX` environment.

3.2 Adding No-cross Constraints

Lazy Constraints

After importing the mathematical model in `CPLEX`, we needed a procedure to generate constraints (22). Consequently, we implemented the *generateNoCross-Constraints* routine as described in Algorithm 1: as can be seen, the addConstraint command is a general one and it does not furnish implementation details. Clearly, this command can be realized with the `CPXaddnewrows` function and we tried this method for comparison purposes.

Nevertheless, `CPLEX` provides also a different technique to append a new constraint to the model. The `CPXaddlazyconstraints` function works in a similar way as `CPXaddnewrows`, yet it should be used to add a large set of constraints that must be satisfied in the final solution but are unlikely to be really violated. In fact, the function creates and appends a constraint to the model only when the solver discovers that this can be actually violated.

input : $turbs$, a vector of cartesian points; $lpModel$, a linear programming model.
output: $lpModel$ extended with no-cross constraints.

for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** n **do**
    for $k \leftarrow 1$ **to** n **do**
      $S \leftarrow \emptyset$
      for $l \leftarrow 1$ **to** n **do**
        if $isCrossing(turbs[i], turbs[j], turbs[k], turbs[l])$ **then**
          $S \leftarrow S \cup \{l\}$
        end
      end
      if $|S| > 0$ **then**
        $lpModel.addConstraint\left(y_{i,j} + \sum_{l \in S} y_{k,l} \leq 1\right)$
      end
    end
  end
end

Algorithm 1: *generateNoCrossConstraints* procedure.

Callbacks

A different technique to discover violated constraints and joining them to the model *on-the-fly* involves the *callbacks*. In general, a callback is a user-defined function X that is passed as a parameter to another function Y. For example, Y could be a general purpose function that utilizes X in order to accomplish

input : $p1$, $p2$, $p3$, $p4$ cartesian points with x and y attributes; $\epsilon > 0$, tolerance parameter.
output: 1 if segments $(p1, p2)$ and $(p3, p4)$ cross each other, 0 otherwise.

$a \leftarrow p1.x - p2.x$
$b \leftarrow p4.x - p3.x$
$c \leftarrow p1.y - p2.y$
$d \leftarrow p4.y - p3.y$
$e \leftarrow p4.x - p2.x$
$f \leftarrow p4.y - p2.y$
$det \leftarrow a \cdot d - b \cdot c$
if $|det| < \epsilon$ **then**
  return 0
end
else
  $\lambda \leftarrow (d \cdot e - b \cdot f) \,/\, det$
  $\mu \leftarrow (a \cdot f - c \cdot e) \,/\, det$
  if $(0 < \lambda < 1) \wedge (0 < \mu < 1)$ **then**
    return 1
  end
  return 0
end

Algorithm 2: *isCrossing* procedure.

specific operations on user-defined data whose details were not available during Y's implementation.

By means of this technique, `CPLEX` allows the user to define functions for his specific purposes and these functions can be called during the optimization process. `CPLEX` provides several kinds of callbacks, nevertheless here we only show how to deal with the ones allowing to add the *lazy constraints*. In the following are reported the steps that should be followed when using callbacks in `CPLEX`.

- **Installing the callback**: this task can be accomplished by using the `CPXsetlazyconstraintcallbackfunc` function that receives as arguments the callback the user has defined and the data needed for its computation (in our specific case, we send the positions of the turbines).
- **Obtaining information**: plenty of useful information can be obtained during the computation by calling the `CPXgetcallbackinfo` routine. For example, by passing the `CPX_CALLBACK_INFO_MY_THREAD_NUM` parameter, the user-defined function can identify the thread in which it is running.
- **Performing specific subroutines:** inside a callback, the user is able to specify any sort of operation she requires for her purposes, remembering that the making use of data provided as a parameter to the callback, must be managed in a thread-safe environment.
- **Adding the cuts**: after generating a useful cut (o more than one), the user can append it to the model by invoking the `CPXcutcallbackadd`

function. The successful production of at least one cut must be notified to `CPLEX` with the following line of code

```
*useraction_p = CPX_CALLBACK_SET;
```

where `useraction_p` is the last parameter in the definition of a callback.

Loop Method

Being a large amount of no-cross constraints our main complication, we can design a simple procedure that appends to the model only the ones really required by the instance. A precise description of our implementation for the so called *Loop Method* is given by Algorithm 3.

The procedure invokes `CPLEX` with an extremely short time limit (30 seconds is generally enough for our testbed) in order to solve the model (initially without no-cross constraints). Subsequently, it checks the solution obtained in the previous step, appending the violated constraints to the model.

These two main steps are iterated until the time limit is reached.

initialize *model* without no-cross constraints
while time limit not reached **do**
  solution $\leftarrow$ solve *model* within a time limit T and get a solution
  if *solution* is an integer feasible solution **then**
    for each $(i, j, k) \in V \times V \times V$ **do**
      $S \leftarrow \emptyset$
      for $l \leftarrow 1$ **to** n **do**
        if $y_{i,j} \in solution$ **and** $y_{k,l} \in solution$ **and**
         $isCrossing(turbs[i], turbs[j], turbs[k], turbs[l])$ **then**
          $S \leftarrow S \cup \{l\}$
        end
      end
      if $|S| > 0$ **then**
        $model.addConstraint\left(y_{i,j} + \sum_{l \in S} y_{k,l} \leq 1\right)$
      end
    end
  end
end

Algorithm 3: The *Loop Method* pseudocode.

4 Heuristic and Matheuristic Solvers

In this chapter, we cover several ideas to solve our original problem nevertheless, these do not guarantee to find an optimal solution. Techniques that make use of *rules of thumb* (specifically designed for a problem), fall under the name of *heuristics*.

4.1 Hard-fixing

A relatively new approach to optimization is provided by *matheuristic algorithms*. Matheuristic is a collection of techniques that combine the power of mathematical solvers with the intuition of heuristic and *metaheuristic* algorithms (see [3]). We investigated one method belonging to this family, named *Hard-fixing*.

After generating the model (11) - (26), the *Hard-fixing* procedure calls `CPLEX` to gain a starting, even infeasible, solution and by means of the slack variables, this is achievable within a small interval of time.

Afterwards, the algorithm examines the y variables and it randomly fixes an arbitrary percentage of such variables. This action is performed by changing the bounds of chosen variables: for example, if a variable is set to 1 in the solution found by `CPLEX`, we change its upper and lower bound to 1 (this can be performed with the `CPXchgbds` function). Subsequently, `CPLEX` is newly called to solve the model thus obtained.

These steps are repeatedly executed until a time limit expires or no improvements can be achieved.

```
initialize model
while time limit not reached do
    solution ← solve model within a time limit T and get a solution
    for i ← 1 to n do
        for j ← 1 to n do
            reset lower and upper bounds in model as 0 ≤ y_{i,j} ≤ 1
            if rand() > FIXING_PROB then
                if y_{i,j} = 0 in solution then
                    set 0 ≤ y_{i,j} ≤ 0 in model
                else
                    set 1 ≤ y_{i,j} ≤ 1 in model
                end
            end
        end
    end
end
```

Algorithm 4: The *Hard-fixing* pseudocode.

4.2 Pure Heuristic Approaches

Branch-and-bound technique and mixed integer linear programming are undoubtedly effective approaches, yet their time complexity is high and unacceptable in certain circumstances. Moreover, the implementation of a practical

MILP solver is usually a task beyond the possibilities of a computer engineer and software like `CPLEX` might be really expensive.

To overcome these problems one can use *heuristic algorithms*, designed to solve a problem in a faster and more efficient fashion than traditional methods by sacrificing optimality for speed.

Heuristics are often used to solve NP-hard problems, for which there is not an efficient way to find a solution in a short time.

Heuristic algorithms can certainly produce suitable solutions, nevertheless, they can also be employed to provide good baselines to feed supplemented optimization algorithms. `CPLEX` is a good example of the latter approach, indeed, it makes available to the user several heuristic routines that can be launched during the computation. The most known and successful of these techniques is the *Relaxation Induced Neighborhood Search* (RINS) [4], a method for exploring the neighbourhood of the current incumbent solution in order to find an improved one.

Below, we describe some auxiliary functions, useful for understanding subsequent algorithms. The pure heuristic routines we have implemented to solve this problem are a *multi-start* (MS) approach and the *Variable Neighborhood Search* (VNS).

Clustering Tree Generation

This function was created in order to avoid starting an optimization routine with a *star graph*, namely a solution in which each turbine is directly connected to the substation. The star graph solution is clearly incorrect because the number of cables entering the substation overcomes our parameter C.

Therefore, we opted to make use of a clustering method to obtain an initial solution that could be closer to our needs.

The algorithm we designed chooses C randomly turbines that become the centres of the clusters in which the graph is subdivided. Then, it scans the remaining turbines and assigns each of them to its nearest centre. The centre of each cluster is directly connected to the substation.

The algorithm pseudocode is reported in Algorithm 5.

input : *turbs*, a vector containing the coordinates of the turbines; C, maximum number of cables the substation can accept.
output: *succ*, a structure containing for each turbine i the *successor* of i.

Initialize *succ* and *visited*, get the index of the substation from *succ* and assign it to *substationIndex*

$succ[substationIndex] \leftarrow substationIndex$
$visited[substationIndex] \leftarrow 1$
for $i \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** n **do**
    $M[i,j] \leftarrow distance(turbs[i], turbs[j])$
  end
end

$S \leftarrow \emptyset$
while $|S| < C$ **do**
  $r \leftarrow$ a random value in $\{1, \ldots, n\} / (S \cup \{substationIndex\})$
  $visited[r] \leftarrow 1$
  $succ[r] \leftarrow substationIndex$
  $S \leftarrow S \cup \{r\}$
end

for $i \leftarrow 1$ **to** n **do**
  if *not* $visited[i]$ **then**
    $index \leftarrow 0$
    $minDist \leftarrow \infty$
    for $j \in S$ **do**
      if $M[i,j] < minDist$ **then**
        $index \leftarrow j$
        $minDist \leftarrow M[i,j]$
      end
    end
    $succ[i] \leftarrow index$
    $visited[i] \leftarrow 1$
  end
end
return *succ*

Algorithm 5: *Clustering Tree Generation* pseudocode.

The twoOpt and threeOpt Routines

In our implementation, the *twoOpt* routine is a local search algorithm that considers every path in the cables routing involving 2 connections and 3 turbines, choosing the solution that provides the best outcome (i.e., the minimum cost).

Actually, this is performed by means of an exhaustive search, examining all the (i, j, k) triples in the V^3 search space and each of these triples matches a path (y_{ij}, y_{jk}). In a similar way, the *threeOpt* algorithm iterates over each tuple $(i, j, k, l) \in V^4$ and evaluates the solutions obtained replacing connections y_{ij} and y_{kl}.

By changing the entries in the *succ* structure, we are able to modify the connections in the cables routing: if we require turbine i (and consequently, all

the turbines in its subtree) to be joined to turbine j, we should perform

$$succ[i] \leftarrow j.$$

This operation guarantees that the number of edges remains $n-1$ (the first entry in the *succ* structure will be always ignored) but it allows the presence of loops. In order to avoid them in the final layout, the evaluation phase assigns high penalties to solutions containing connections with cycles.

Both *twoOpt* and *threeOpt* return explicitly the value of the best solution found and implicitly the *succ* structure matching such a solution.

The naming rule adopted by these routines was generated during the development phase. In fact, according to this rule, *oneOpt* is the most basic local search approach that can be accomplished, and it corresponds to modify the successor of each node, attempting to connect it to all the other turbines.

This process can be easily generalized by producing a *k-Opt* routine that will generate n^{k+1} different solutions. Nevertheless, even if this number is not high, it should be remembered that each solution thus found must be evaluated.

Solution Evaluation

Clearly, the most onerous operation in the *twoOpt* and *threeOpt* routines is the evaluation of the new solutions found or, in other words, the evaluation of the *succ* structure according to our mathematical model. In order to achieve this task, we design an algorithm that accomplishes as fewer activities as possible.

The algorithm starts verifying if the solution is rightly connected and if the edges do not form any cycles. Then it checks the flows in every edge, adding a penalty to the solution score for each flow that does not respect the constraint imposed by its edge. Afterwards, the cables entering the substation are scanned while the last and harder step is the control of the no-cross constraints (22).

In order to avoid unnecessary operations, the value of the best solution found until that moment, is provided to the function computing the evaluation. In this way, the evaluation can be arrested by several checkpoints located in strategic places, if the current score overcomes the best solution value.

Multi-Start

The first heuristic we tested on the cables routing problem was a *multi-start* approach. This is a quite straightforward method however, it was able to produce interesting results.

The procedure repeatedly tries to make small improvements until a local optimum point is reached. Suddenly the best solution is preserved and a new starting basis is provided by the *clusteringTreeGeneration* function with a different random seed, in order to produce a distinct basis. The entire procedure terminates when a user-defined time limit is reached. The algorithm pseudocode is reported in Algorithm 6.

When the maximum number of cables entering the substation is not provided (i.e., this is infinite), a value proportional to the number of turbines can be chosen, however, for these cases, we set this parameter to 10.

Thanks to the chaotic structure of the problem, small changes in the starting cables routing can produce totally different outcomes and although an optimal

solution is hardly reachable, providing a suitable initial base, usually enables to quickly generate feasible cables routings.

input : *turbs*, a vector containing the coordinates of the turbines; C, the maximum number of cables the substation can accept; *succ*, an empty structure; the starting random *seed*.
output: *minCost*, the value of the best solution found; (implicitly) *succ*, a structure containing the best solution found.

initialize structure *bestSucc*
$minCost \leftarrow \infty$
$iteration \leftarrow 0$
while time limit not reached **do**
  $seed \leftarrow seed + iteration$
  set the system random seed to *seed*
  $succ \leftarrow clusteringTreeGeneration(turbs, C)$
  $improvement \leftarrow 1$
  while improvement **do**
    $cost \leftarrow twoOpt(succ, minCost)$
    $improvement \leftarrow 0$
    if $cost < minCost$ **then**
      copy *succ* to *bestSucc*
      $improvement \leftarrow 1$
      $minCost \leftarrow cost$
    end
  end
end
copy *bestSucc* to *succ*
return *minCost*

Algorithm 6: *Multi-start* approach pseudocode.

Variable Neighborhood Search

This heuristic algorithm bases its strategy on the local search: it first discovers a local optimum point and then upsets this solution in order to escape from such a point and try to reach the global optimum one.

In the variant of the procedure we designed, we start from an initial solution given by the *Dijkstra algorithm* (a description of this algorithm can be found in [7]), such a solution is then provided to the VNS routine. The VNS changes one connection at a time, saving it whenever improvements are encountered. When no further advancements can be found, the *threeOpt* routine is launched, trying to escape from the local optimum point.

The solution obtained with the Dijkstra algorithm can be randomized to obtain different starting basis and repeat the entire procedure in order to achieve different results. The way we modified the Dijkstra algorithm is extremely simple. This algorithm bases its choice on the costs of the edges hence, we change the cost of each edge proportionally to its length, by adding a random perturbation to increment or decrement it.

input : *succ*, a structure containing for each turbine i the *successor* of i.
output: *minCost*, the value of the best solution found; (implicitly) *succ*, a structure containing the best solution found.

```
minCost ← ∞
while 1 do
    currentMinCost ← minCost
    for i ← 1 to n do
        for j ← 1 to n do
            tmp ← succ[i]
            succ[i] ← j
            cost ← evaluate(succ)
            if cost < currentMinCost then
                currentMinCost ← cost
            else
                succ[i] ← tmp
            end
        end
    end
    if currentMinCost < minCost then
        comment: local optimum not reached, update cost and repeat
          the previous routine
        minCost ← currentMinCost
    else
        comment: local optimum reached
        cost ← threeOpt(succ)
        if minCost ≤ cost then
            return minCost
        end
    end
end
```

Algorithm 7: *Variable Neighborhood Search* pseudocode.

5 Results

The real world cases we considered for this study were obtained from the wind farms listed in Table 1. Assigning them different sets of cables, we produced the testbed for our experiments with 24 instances as reported in Table 2.

In addition to the original cables (*cb01* to *cb05*) denominated CAPEX (CAPital EXpenditure), since these are only considered as an immediate expense, the authors of the article [6] developed other sets of cables estimating future costs. In this way, for each of the original sets of cable (now reporting the *capex* suffix), a new one was obtained in order to compare the different solutions.

We carried out our experiments on a personal computer equipped with an Intel i3 CPU 530, 2.93 GHz x4, 7.7 GB RAM memory and the `IBM ILOG CPLEX 12.8` software.

Name	Site	n. of turbines	C	Allowed cables
wf01	Horns Rev 1	80	10	cb01-cb02-cb05
wf02	Kentish Flats	30	∞	cb01-cb02-cb03-cb04-cb05
wf03	Ormonde	30	4	cb03-cb04
wf04	DanTysk	80	10	cb01-cb03-cb04-cb05
wf05	Thanet	100	10	cb04-cb05

Table 1: The real wind farms we considered (from [6]).

The results were obtained by running each instance with a 3600 seconds time limit. Therefore, these have been compared with the results produced through other similar techniques or different configurations.

Consequently, we do not record the values of infeasible solutions and we report beside each result the error as compared to the best lower bound known (the optimal solution found for the same instance by Fischetti and Pisinger in [6]). This error is calculated as

$$\left(1 - \frac{LB}{val}\right) \cdot 100.$$

In Table 3, we reported the data obtained generating all the constraints (22) by means of the *lazy constraints* and the *new rows*. As it can be seen, the model (11) - (22) without slack variables can be used to solve the smaller cases, however the problem becomes unmanageable in cases with 80 or 100 turbines (the software was sometimes stopped by the operating system before reaching the time limit).

The main difference in using the *lazy constraints* instead of the *new rows* is that the former results to be much faster than the latter. This fact can be observed in instances 9, 12 and 14.

Through these examples, we can also notice that `CPLEX` stops the computation after finding the optimal solution according to the model and the data we provided. Nevertheless, the gap between the solution we found and the solution in [6] is not exactly zero due to internal `CPLEX`'s tolerances.

Inst.	Wind farm	Cable set
1	wf01	wf01_cb01_capex
2		wf01_cb01
3		wf01_cb02_capex
4		wf01_cb02
5		wf01_cb05_capex
6		wf01_cb05
7	wf02	wf02_cb01_capex
8		wf02_cb01
9		wf02_cb02_capex
10		wf02_cb02
11		wf02_cb03
13		wf02_cb04
14	wf02	wf02_cb05_capex
15		wf02_cb05
16	wf03	wf03_cb03_capex
17		wf03_cb03
18		wf03_cb04_capex
19		wf03_cb04
20	wf04	wf04_cb01_capex
21		wf04_cb01
26	wf05	wf05_cb04_capex
27		wf05_cb04
28		wf05_cb05_capex
29		wf05_cb05

Table 2: The testbed considered in this report.

Inst.	Lazy constraints			New rows		
	Time	bestsol	% err.	Time	bestsol	% err.
1	3600.75	-	-	3602.66	-	-
2	3602.65	-	-	3600.00	-	-
3	3600.77	-	-	3604.17	-	-
4	3602.80	-	-	3600.00	-	-
5	3600.65	-	-	3600.00	-	-
6	3603.39	-	-	3600.00	-	-
7	3600.04	8555171.40	0.00	3602.21	8555171.40	0.00
8	3600.57	8806839.00	0.00	3600.08	8806839.00	0.00
9	4.30	11159487.98	9.88	1903.35	11159487.98	9.88
10	3600.04	10303320.45	0.00	3600.11	10303320.45	0.00
12	8.07	9009167.55	4.49	3409.50	9009167.55	4.49
13	706.51	8933494.43	0.00	1298.37	8933494.43	0.00
14	39.05	10377051.63	1.96	3091.89	10377051.63	1.96
15	3600.06	10348430.55	0.00	3600.11	10348430.55	0.00
16	3600.06	8054844.90	0.00	3600.11	8054844.90	0.00
17	3600.08	8560008.61	0.00	3600.09	8560008.61	0.00
18	3600.07	8357195.91	0.00	3600.10	8357195.91	0.00
19	3600.07	9178499.87	0.00	3600.13	9178499.87	0.00
20	3600.72	-	-	3603.23	-	-
21	3605.56	-	-	3600.00	-	-
26	10.95	-	-	365.78	-	-
27	3604.24	-	-	3612.16	-	-
28	11.15	-	-	290.77	-	-
29	3607.85	-	-	3600.00	-	-

Table 3: Results comparison for *lazy constraints* and *new rows*, not involving callbacks.

Differently from what can seen above, the relaxation of the model implies a general slowdown in the computation (Table 4).

Nevertheless, acceptable solutions can be calculated within the time limit for the *Horns Rev 1* wind farm (which counts 80 turbines) and for the Thanet wind farm (instance 27, with 100 turbines).

Inst.	Lazy constraints			New rows		
	Time	bestsol	% err.	Time	bestsol	% err.
1	3601.45	19624392.92	0.96	3603.21	19549735.53	0.58
2	3601.29	21509919.98	0.50	3603.43	21486462.37	0.39
3	3600.99	23120440.28	2.20	3603.60	22616959.38	0.02
4	3601.45	25316975.42	3.44	3603.74	24931979.39	1.95
5	3600.97	24505448.34	4.17	3602.86	23631255.96	0.63
6	3601.59	24877152.27	0.44	3603.48	25349299.98	2.29
7	3600.06	8555171.40	0.00	3600.10	8555171.40	0.00
8	3600.08	8806839.00	0.00	3600.10	8806839.00	0.00
9	2052.98	11159487.98	9.88	3600.09	11159487.98	9.88
10	3600.07	10303320.45	0.00	3600.09	10303320.45	0.00
12	3600.06	9009167.55	4.49	3600.11	9009167.55	4.49
13	3600.08	8933494.43	0.00	1633.34	8933494.43	0.00
14	3600.05	10377051.63	1.96	3600.12	10377051.63	1.96
15	3600.09	10348430.55	0.00	3600.14	10348430.55	0.00
16	3600.06	8054844.90	0.00	3600.09	8054844.90	0.00
17	3600.08	8560008.61	0.00	3600.14	8560008.61	0.00
18	3600.05	8357195.91	0.00	3600.10	8357195.91	0.00
19	3600.09	9178499.87	0.00	3600.11	9178499.87	0.00
20	3601.10	38977593.84	0.00	3603.72	39779972.34	2.02
21	3601.02	-	-	3600.00	-	-
26	1343.32	-	-	1430.88	-	-
27	3602.63	23519984.24	0.67	3608.63	23929413.92	2.37
28	2996.94	-	-	1164.88	-	-
29	3602.84	-	-	3607.03	-	-

Table 4: Results comparison for *lazy constraints* and *new rows* with the relaxed constraints (23) and (25), not involving callbacks.

Being the dynamic memory allocation one of the most onerous operations that can be performed during a software execution, we investigated how this impacts on callback performances.

Indeed, the callback we implemented needs auxiliary data structures in order to accomplish its task. Therefore we developed a method to allocate all the dynamic memory required before the computation starts, ensuring this was thread-safe.

As it can be observed in Table 5, there is no evidence whether the multi-thread optimization for the callbacks could be considered faster than normal callbacks. In order to confirm this point, we ran several experiments with a shorter time limit (600 seconds) on instances we considered meaningful (1, 3, 5 and 27). Each experiment was repeated 5 times, and the average values we collected are reported in Table 6.

Moreover, we replicated the tests by way of using the look-up table for the no-cross constraints (Table 7).

Inst.	Callback			Callback (multi-thread optimized)		
	Time	bestsol	% err.	Time	bestsol	% err.
1	3600.10	19538324.05	0.52	3600.12	19545575.16	0.56
2	3600.41	21462015.17	0.27	3600.46	21499344.29	0.45
3	3600.09	22679204.17	0.30	3600.12	22617203.57	0.02
4	3600.34	24508837.24	0.26	3600.34	24452244.24	0.03
5	3600.06	23957757.83	1.98	3600.09	23693555.56	0.89
6	3600.49	24856351.30	0.35	3600.37	25102530.16	1.33
7	3600.02	8555171.40	0.00	3600.02	8555171.40	0.00
8	3600.04	8806839.00	0.00	3600.03	8806839.00	0.00
9	3600.02	11159487.98	9.88	2112.68	11159487.98	9.88
10	3600.03	10303320.45	0.00	3600.03	10303320.45	0.00
12	3600.01	9009167.55	4.49	3600.02	9009167.55	4.49
13	3600.05	8933494.43	0.00	3600.04	8933494.43	0.00
14	3600.01	10377051.63	1.96	3600.01	10377051.63	1.96
15	3600.03	10348430.55	0.00	3600.04	10348430.55	0.00
16	3600.02	8054844.90	0.00	3600.02	8054844.90	0.00
17	3600.04	8560008.61	0.00	3600.04	8560008.61	0.00
18	3600.03	8357195.91	0.00	3600.02	8357195.91	0.00
19	3600.03	9208903.51	0.33	3600.04	9208903.51	0.33
20	3600.14	43285518.01	9.95	3600.15	38977593.84	0.00
21	3600.21	-	-	3600.79	-	-
26	1664.10	-	-	2497.38	-	-
27	3600.57	23521294.00	0.68	3601.13	23532718.33	0.73
28	3600.12	-	-	3600.22	-	-
29	3600.34	-	-	3600.81	-	-

Table 5: Results comparison for the solver using the callbacks (*left*) and employing the multi-thread optimized callbacks (*right*).

We can conclude that the results do not confirm our initial hypotheses. In fact, we thought that the initial allocation of all the necessary memory and a look-up table could bring significant speed-ups.

We explain this fact by supposing that `CPLEX` does not utilize the callbacks enough to produce substantial differences between the alternatives we proposed.

Inst.	Callback			Callback (multi-thread optimized)		
	Time	bestsol	% err.	Time	bestsol	% err.
1	600.11	20662980.97	5.91	600.13	19908107.50	2.34
3	600.09	23016288.32	1.75	600.14	23012144.75	1.72
5	600.07	25193374.82	6.79	600.08	25733347.81	8.63
27	600.55	23598627.24	1.00	601.16	23899749.02	2.23

Table 6: (Average on 5 experiments with different random seeds). Results comparison in performance for the solver using the callbacks (*left*) and employing the multi-thread optimized callbacks (*right*).

Inst.	Callback			Callback (look-up table)		
	Time	bestsol	% err.	Time	bestsol	% err.
1	600.13	19811369.14	1.89	600.75	19899030.29	2.32
3	600.11	22959390.16	1.51	600.83	23065770.03	1.97
5	600.09	25703741.98	8.64	600.72	26346597.73	10.87
27	601.04	24512033.73	4.69	602.34	24235865.08	3.61

Table 7: (Average on 5 experiments with different random seeds). Results comparison in performance for the solver employing the callbacks (*left*) and employing the callbacks with the look-up table for the no-cross constraints (*right*).

Inst.	*Loop Method* iteration time 30 s			*Loop Method* iteration time 60 s		
	Time	bestsol	% err.	Time	bestsol	% err.
1	3600.07	19509111.10	0.37	3600.08	19521220.02	0.43
2	3600.12	22153300.29	3.39	3600.15	22062434.37	2.99
3	3600.06	22654673.18	0.19	3600.07	22627228.74	0.07
4	3600.10	26977790.42	9.39	3600.12	25957849.97	5.83
5	3600.04	23791669.01	1.30	3600.05	23804774.66	1.35
6	3600.15	25102631.11	1.33	3600.18	24928381.90	0.64
7	3600.01	8555171.40	0.00	3600.01	8555171.40	0.00
8	3600.02	8806839.00	0.00	3600.02	8806839.00	0.00
9	3600.01	11159487.98	9.88	3600.01	11159487.98	9.88
10	3600.02	10303320.45	0.00	3600.02	10303320.45	0.00
12	3600.01	9009167.55	4.49	3600.01	9009167.55	4.49
13	3600.02	8933494.43	0.00	3600.02	8933494.43	0.00
14	3600.01	10377051.63	1.96	3600.01	10377051.63	1.96
15	3600.02	10348430.55	0.00	3600.02	10348430.55	0.00
16	3600.01	8054844.90	0.00	3600.01	8054844.90	0.00
17	3600.02	8560008.61	0.00	3600.02	8560008.61	0.00
18	3600.01	8357195.91	0.00	3600.01	8357195.91	0.00
19	3600.02	9208903.51	0.33	3600.02	9208903.51	0.33
20	3600.05	40207339.57	3.06	3600.05	-	-
21	3600.09	-	-	3600.12	45220170.96	0.80
26	3600.06	-	-	3600.08	-	-
27	3600.20	24058864.39	2.90	3600.22	24020158.41	2.74
28	3600.05	-	-	3600.06	-	-
29	3600.12	-	-	3600.14	-	-

Table 8: Result comparison for the relaxed model using the *Loop Method* with different iteration times.

We launched the *Loop Method* with different iteration times (the maximum time `CPLEX` elapses inside each loop). The data we collected in Table 8 do not show major differences but surprisingly, for the first time, `CPLEX` was able to find a quite optimal solution for the instance 21.

Inst.	*Hard-fixing* iteration time 30 s			*Hard-fixing* iteration time 60 s		
	Time	bestsol	% err.	Time	bestsol	% err.
1	3600.05	19632158.38	1.00	3600.05	19659116.23	1.13
2	3600.11	21553383.25	0.70	3600.12	21547248.31	0.67
3	3600.04	23986919.48	5.73	3600.05	22922150.76	1.35
4	3600.08	24832342.52	1.56	3600.10	25034512.98	2.35
5	3600.04	23650237.11	0.71	3600.03	25249971.32	7.00
6	3600.11	24929118.77	0.64	3600.12	24833083.68	0.26
7	39.33	8555171.40	0.00	69.29	8555171.40	0.00
8	60.06	8806839.00	0.00	106.65	8806839.00	0.00
9	30.36	11159487.98	9.88	60.36	11159487.98	9.88
10	53.19	10303320.45	0.00	83.25	10303320.45	0.00
12	40.19	9009167.55	4.49	70.18	9009167.55	4.49
13	31.66	8933494.43	0.00	61.66	8933494.43	0.00
14	40.11	10377051.63	1.96	61.26	10377051.63	1.96
15	39.58	10352340.09	0.04	86.69	10348430.55	0.00
16	90.05	8073836.94	0.24	180.05	8073836.94	0.24
17	60.06	8560008.61	0.00	120.07	8560008.61	0.00
18	90.05	8422769.68	0.78	139.79	8357195.91	0.00
19	60.08	9263066.88	0.91	120.07	9263066.88	0.91
20	3600.05	-	-	3600.04	-	-
21	3600.07	45421513.36	1.24	3600.08	-	-
26	3600.16	-	-	3600.53	-	-
27	3600.75	-	-	3600.09	-	-
28	3600.05	-	-	3600.05	-	-
29	3600.14	-	-	3600.10	-	-

Table 9: Result comparison for the relaxed model using the *Hard-fixing* approach with different iteration times and 0.5 fixing probability.

The *Hard-fixing* offers two main parameters to analyze: the iteration time (as in the *Loop Method*) and the *fixing probability*, namely the probability to constrain a variable representing a cable.

If we start considering the iteration time (Table 9), it becomes evident that the *Hard-fixing* is the best approach for the smaller instances, providing optimal solutions in a very short time. There are no notable differences if we change this parameter and the most interesting aspect is that the number of iterations does not seem to vary (see instances 16, 17, 19).

Inst.	*Hard-fixing* prob 0.3			*Hard-fixing* prob 0.7		
	Time	bestsol	% err.	Time	bestsol	% err.
1	3600.05	19479436.83	0.22	210.91	20945481.84	7.20
2	3600.11	21517678.64	0.53	3600.12	21534815.61	0.61
3	3600.04	22695422.11	0.37	402.89	23489037.16	3.73
4	3600.07	25699635.05	4.88	3600.07	25452220.12	3.95
5	3600.03	24300076.66	3.36	90.69	-	-
6	3600.11	24845955.65	0.31	3600.14	24790222.64	0.09
7	60.04	8555171.40	0.00	30.43	8555171.40	0.00
8	60.06	8819813.40	0.15	30.41	8819813.40	0.15
9	32.76	11435665.94	12.06	30.14	11159487.98	9.88
10	60.06	10303320.45	0.00	30.95	10303320.45	0.00
12	60.04	9009167.55	4.49	30.12	9009167.55	4.49
13	53.98	8933494.43	0.00	30.35	8933494.43	0.00
14	60.05	10377051.63	1.96	30.13	10377051.63	1.96
15	60.07	10348430.55	0.00	30.29	10348430.55	0.00
16	60.05	8054844.90	0.00	42.71	8054844.90	0.00
17	60.11	8560008.61	0.00	60.07	8560008.61	0.00
18	60.04	8635776.60	3.23	60.05	8635776.60	3.23
19	60.07	9267044.29	0.96	60.07	9267044.29	0.96
20	932.04	-	-	3600.05	-	-
21	391.88	-	-	3600.03	-	-
26	3600.07	-	-	3600.07	-	-
27	3600.10	23667734.80	1.29	3600.04	-	-
28	3600.07	-	-	3600.06	-	-
29	1929.72	-	-	3600.13	-	-

Table 10: Result comparison for the relaxed model using the *Hard-fixing* approach with different fixing probabilities and iteration time of 30 seconds.

As we can see from Table 10, by increasing the fixing probability the algorithm performs less iterations (instances 7, 8), however, this does not guarantee better outcomes (as for instances 1, 3).

In order to provide a final overview of all the approaches using `CPLEX`, we gathered the previous data in Table 11 and, for each instance, we reported the method that achieved the best outcome. Instances from 7 to 19 report the $*$ symbol, since many or all approaches were able to find the optimal solution. In the other cases, there is only one big instance (instance 20 counting 80 turbines) for which two methods obtained an error equal to zero.

Considering the wind farms with 80 turbines or more, the algorithms we studied and implemented, found results very close to the best outcomes in 9 instances over 12 (in the remaining cases we only found infeasible solutions). The most remarkable aspect we noticed is that there is no one method *to rule them all*, but the key concept is to attempt many different approaches.

Inst.	method	bestsol	% err.	Fischetti & Pisinger
1	Hard-fixing $p = 0.3, t = 30$	19479436.83	0.22	19436700.18
2	Callback	21462015.17	0.27	21403410.11
3	Newrows	22616959.38	0.02	22611988.67
4	Callback (multithread)	24452244.24	0.03	24445688.02
5	Newrows	23631255.96	0.63	23482483.25
6	Hard-fixing $p = 0.7, t = 30$	24790222.64	0.09	24768927.72
7	*	8555171.40	0.00	8555171.40
8	*	8806839.00	0.00	8806838.99
9	*	11159487.98	9.88	10056670.31
10	*	10303320.45	0.00	10303320.51
12	*	9009167.55	4.49	8604208.93
13	*	8933494.43	0.00	8933494.59
14	*	10377051.63	1.96	10173931.59
15	*	10348430.55	0.00	10348430.63
16	*	8054844.90	0.00	8054844.90
17	*	8560008.61	0.00	8560008.68
18	*	8357195.91	0.00	8357195.91
19	*	9178499.87	0.00	9178499.88
20	Lazy constraints - Callback (multithread)	38977593.84	0.00	38977593.84
21	Loop Method	45220170.96	0.80	44857986.73
26	-	-	-	22337935.84
27	Lazy constraints	23519984.24	0.67	23362025.61
28	-	-	-	26637602.25
29	-	-	-	27295289.87

Table 11: Final result comparison with Fischetti and Pisinger [6] best solutions.

A separate description is required for the heuristic algorithms we developed. From the data that we collected in Table 12, it is easy to notice that the outcomes of these approaches are generally worse than the previous ones.

In fact, developing a suitable heuristic for a specific optimization problem, requires much more time and trials than producing programs for a solver. Clearly, this way is feasible in the case the solver is available and once one has learned the basic notions of linear and integer optimization.

As it can be observed, the VNS algorithm came generally close to the best results for the smaller instances (7 to 19) while the MS approach found feasible solutions in the first wind farm (instances 1 to 6).

We also provided the intermediate values founded by the VNS algorithm while running on instances 3 and 8 (the former with a 3600 seconds time limit, the latter with 600 seconds one). These results are shown in Figure 2 and Figure 3: in these pictures, the red line indicates the minimum value obtained in the experiment.

The value of the solutions found changes rapidly in the second case because it is simple for the VNS to evaluate it, reaching a local optimum point and repeating the process. The situation is completely different in the first case in which, even considering a wider time interval, it becomes harder for the VNS to find new solutions.

Inst.	VNS		MS		Fischetti & Pisinger
	bestsol	% err.	bestsol	% err.	bestsol
1	22198897.67	12.44	20514419.34	5.25	19436700.18
2	23317581.97	8.21	23660016.06	9.54	21403410.11
3	27255268.38	17.04	25759529.15	12.22	22611988.67
4	28266065.62	13.52	27491342.61	11.08	24445688.02
5	31661232.06	25.83	27482670.99	14.56	23482483.25
6	-	-	26330480.07	5.93	24768927.72
7	8562852.92	0.09	8588547.52	0.39	8555171.40
8	8809664.96	0.03	8819918.29	0.15	8806838.99
9	11949759.75	15.84	13040367.45	22.88	10056670.31
10	10303477.14	0.00	10554806.92	2.38	10303320.51
12	9609993.31	10.47	9378158.57	8.25	8604208.93
13	8933494.43	0.00	9771466.67	8.58	8933494.59
14	11069102.24	8.09	10802067.47	5.81	10173931.59
15	10348430.55	0.00	10392104.55	0.42	10348430.63
16	8054844.90	0.00	8499654.19	5.23	8054844.90
17	8560008.61	0.00	8676395.29	1.34	8560008.68
18	8357195.91	0.00	10439670.97	19.95	8357195.91
19	9178499.87	0.00	11478140.50	20.03	9178499.88
20	56698315.12	31.25	-	-	38977593.84
21	63237214.55	29.06	-	-	44857986.73
26	-	-	-	-	22337935.84
27	26068127.29	10.38	-	-	23362025.61
28	-	-	-	-	26637602.25
29	-	-	-	-	27295289.87

Table 12: VNS and *multi-start* with 3600 seconds time limit compared to best solutions in [6].

Finally, in order to better understand what the algorithms were producing, we developed a program to plot the cable routing once we had obtained a solution. This program was developed by way of using the Python programming language [11], Matplotlib [9] and NetworkX [8]. The latter is a resourceful software library written for Python, particularly useful when dealing with graphs.

In Figures from 4 to 9, we can observe how our solutions appear. The turbines are represented with a green dot, the substation with a red one and the numbers over the connections indicate the maximum flow they are able to transfer.

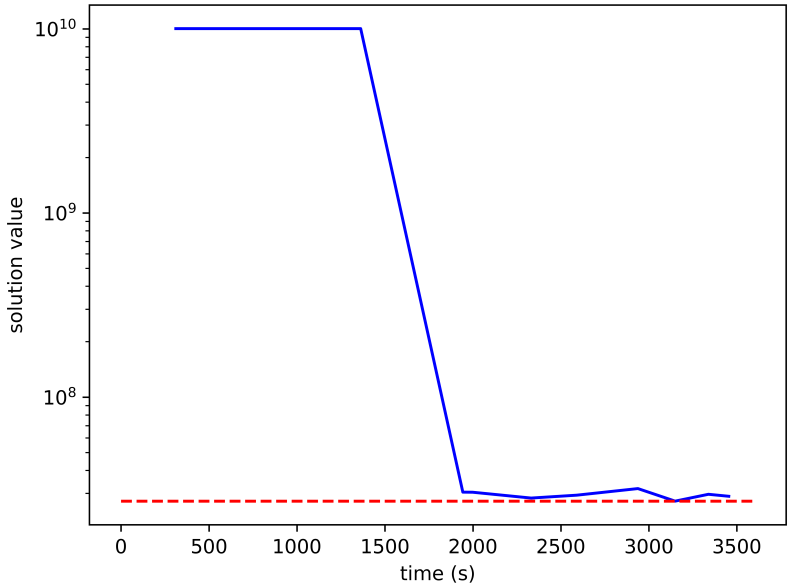

Figure 2: Intermediate values found by the VNS algorithm for instance 3 with a 3600 seconds time limit.

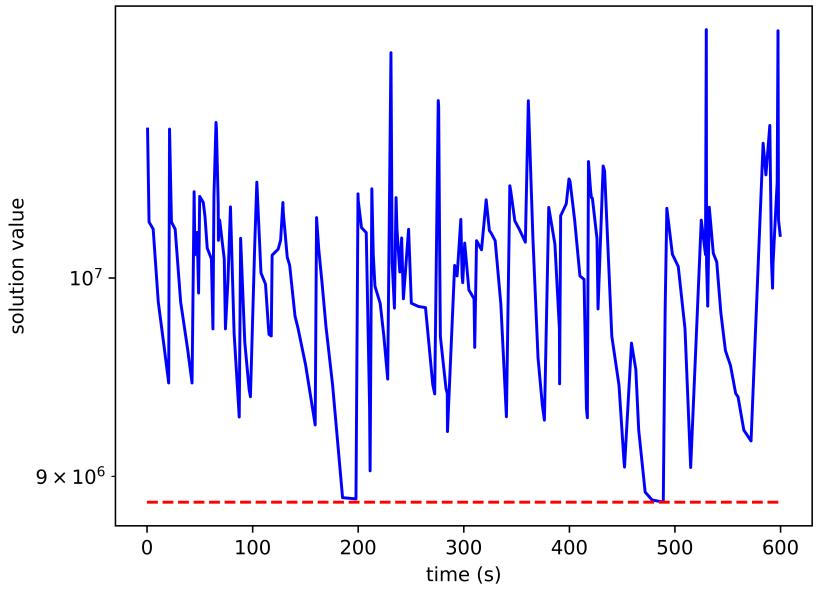

Figure 3: Intermediate values found by the VNS algorithm for instance 8 with a 600 seconds time limit.

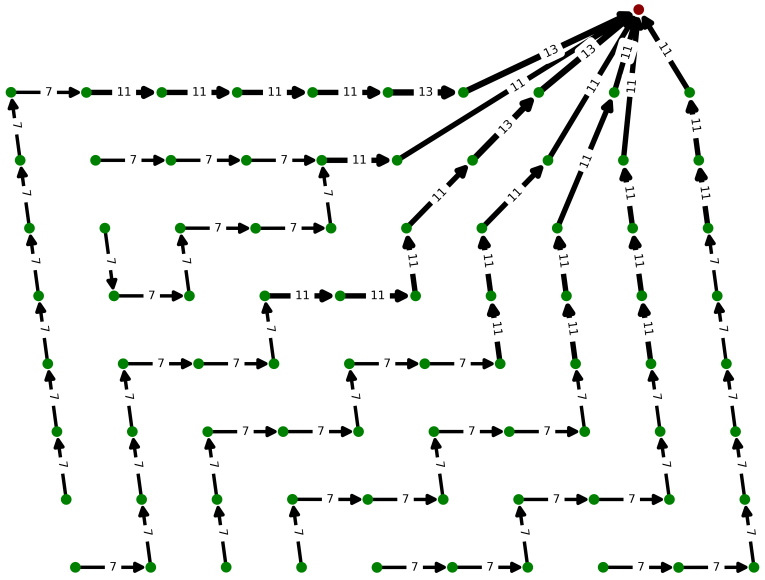

Figure 4: Final layout for instance 1 obtained with *Hard-fixing* ($p = 0.3, t = 30\,s$).

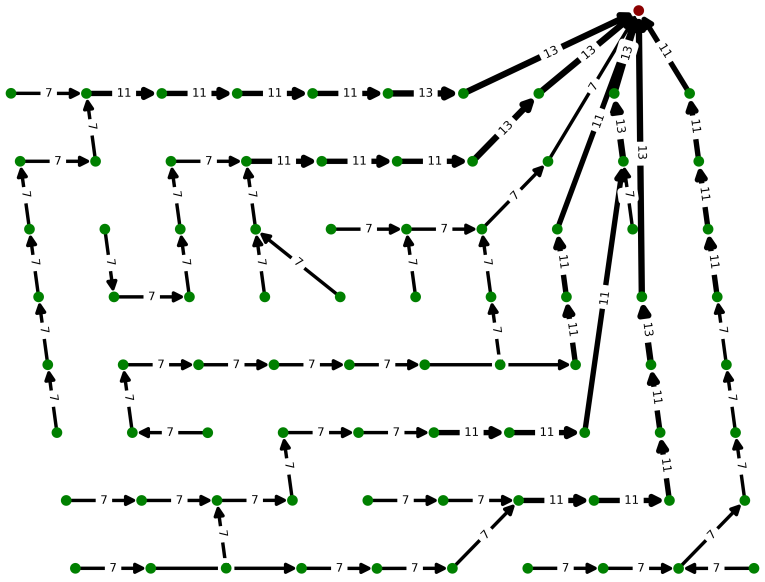

Figure 5: Final layout for instance 1 obtained with *Hard-fixing* ($p = 0.7, t = 30\,s$).

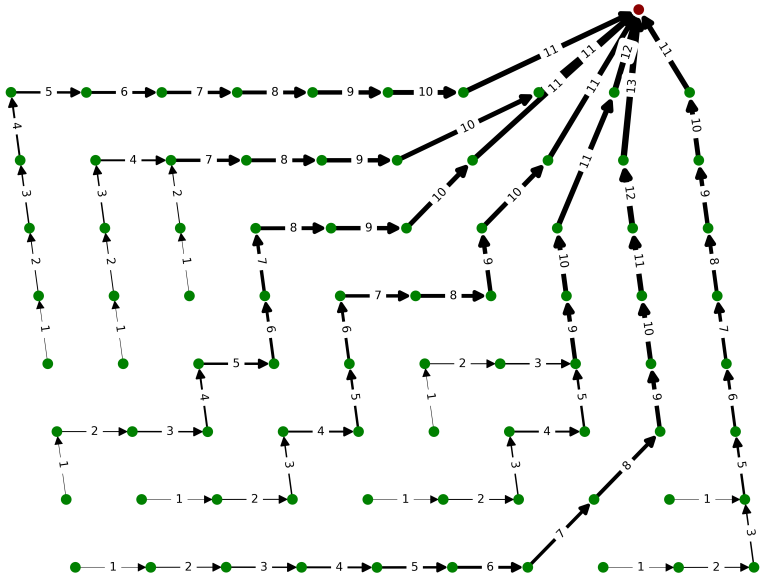

Figure 6: Final layout for instance 2 obtained with *Hard-fixing* ($p = 0.3, t = 30\,s$).

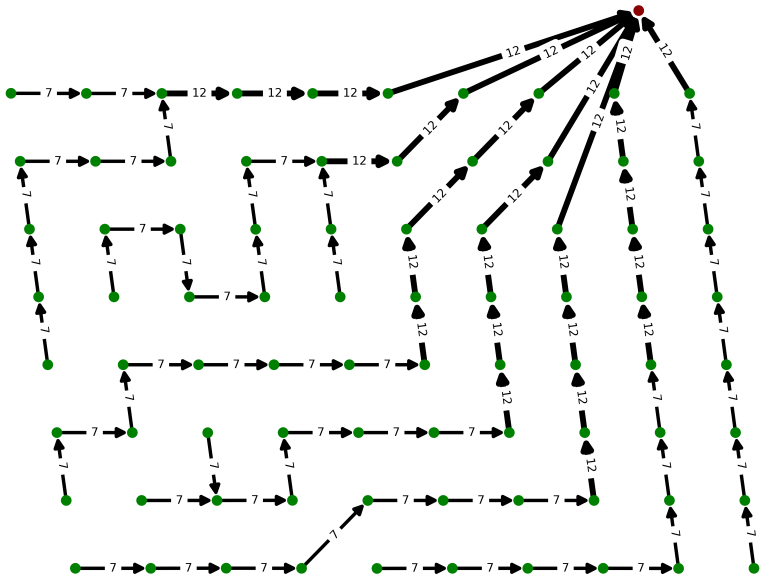

Figure 7: Final layout for instance 3 obtained with *Hard-fixing* ($p = 0.3, t = 30\,s$).

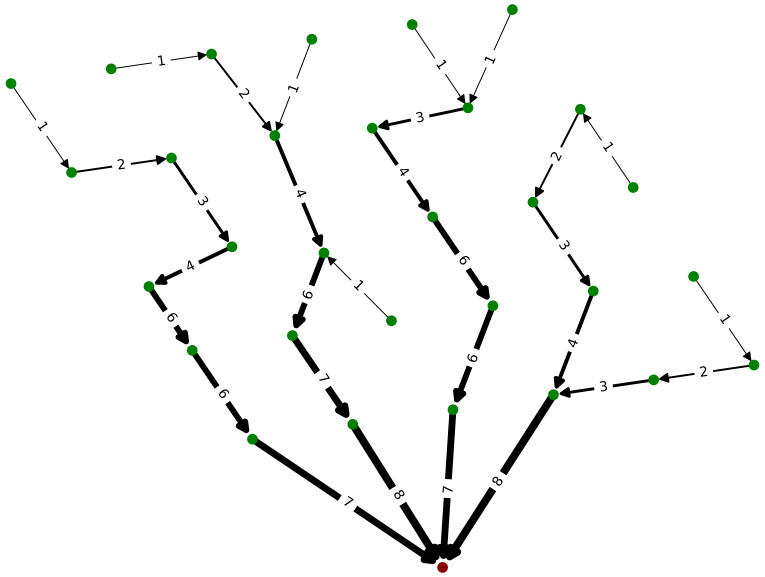

Figure 8: Final layout for instance 10 obtained with *Hard-fixing* ($p = 0.3, t = 30\,s$).

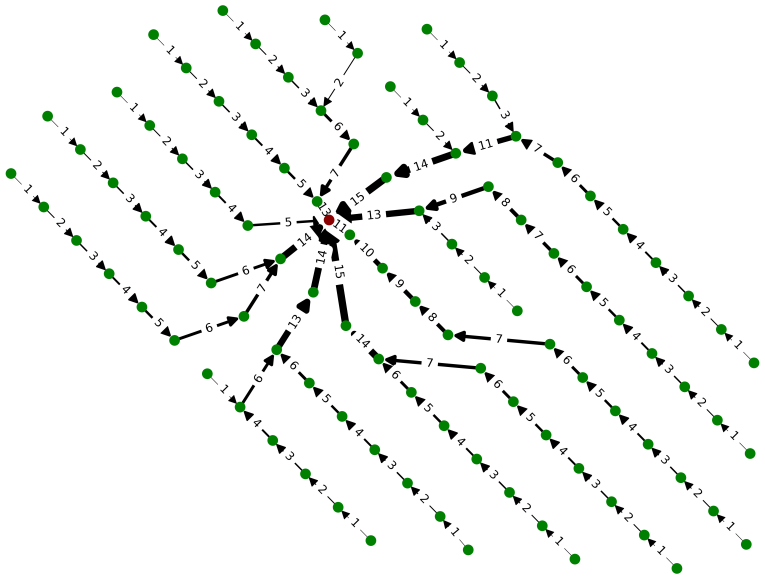

Figure 9: Final layout for instance 27 obtained with *Hard-fixing* ($p = 0.3, t = 30\,s$).

6 Conclusions

This project allowed us to engage with common issues in the practical optimization field: from the comprehension and the evaluation of a mathematical model to the implementation of integer programming techniques.

The first approach with `CPLEX` was rather challenging, nevertheless, we shortly developed the abilities and tools to ensure that the programs we were writing acted as we expected from the theory.

Once we had reached a high level of knowledge about the problem that we were attempting to solve, we have dealt with the development of heuristic algorithms. Such an approach has conferred us a larger amount of freedom but also more responsibilities. Indeed, it was easier to produce new ideas, yet we ultimately came to realize that many of these were wrong.

Moreover, we discovered how a solver can impact in a real-world project in terms of cost saving and how complex it is to achieve similar results by way of implementing specific algorithms for the task.

We think that one of the main purposes of this course is to acquire a deeper critical view of software development in optimization. By merely studying the theoretical aspects of the discipline, one cannot state which is a *large* instance, and how much time is required to produce an optimal solution, or whether an approach is suitable for a given instance.

Appendix A

Setting CPLEX on Mac OS and Ubuntu

Here we provide a procedure for installing the `IBM ILOG CPLEX Optimization Studio` software on a Mac OS or a Ubuntu machine.

The `CPLEX` installer is distributed as a *.bin* file and it can be easily downloaded from `https://www.ibm.com/it-it/marketplace/ibm-ilog-cplex`. Students and academics can easily obtain a full version of the product, on the other cases, only a demo version is available.

In order to configure `CPLEX` on a Mac OS machine, we first have to assign the right permission to execute the installation file. We can make it with the command

```
chmod u+x CPLEX_OPT_STUD_[version]_FOR_OSX.bin
```

Then we can install the software running

```
./CPLEX_OPT_STUD_[version]_FOR_OSX.bin
```

The procedure is quite similar for Ubuntu machines. It should look like as the following

```
chmod u+x cplex_studio[version].linux-x86.bin
./cplex_studio[version].linux-x86.bin
```

To be certain the installation is correctly accomplished, we can run some examples, such as

```
cd /CPLEX_BASE_DIR/examples/SYSTEM/static_pic
make ilolpex1
./ilolpex1 -r
```

By using `CPLEX` with the C language, as we did, it is necessary to tell the compiler how to find the *include* directory and to specify the linker where is the library location. With the `gcc` compiler, this should look like as

```
gcc -I/CPLEX_BASE_DIR/include/ilcplex
    -L/CPLEX_BASE_DIR/lib/x86-64_[SYSTEM]/static_pic
    -lcplex -lm -lpthread -ldl
```

To conclude, it shouldn't forget to add this line to the file in which `CPLEX` functions are called

```
#include <cplex.h>
```

References

[1] Cplex lp file format. 2018-12-04, `http://lpsolve.sourceforge.net/5.0/CPLEX-format.htm`.

[2] Ibm ilog cplex optimization studio. 2018-12-04, `https://www.ibm.com/analytics/cplex-optimizer`.

[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, 2009. Springer Berlin Heidelberg.

[4] Emilie Danna, Edward Rothberg, and Claude Le Pape. Exploring relaxation induced neighborhoods to improve mip solutions. *Mathematical Programming*, 102(1):71–90, 2005.

[5] Martina Fischetti, Matteo Fischetti, and Michele Monaci. Optimal turbine allocation for offshore and onshore wind farms. In Katsuki Fujisawa, Yuji Shinano, and Hayato Waki, editors, *Optimization in the Real World*, pages 55–78, Tokyo, 2016. Springer Japan.

[6] Martina Fischetti and David Pisinger. Optimizing wind farm cable routing considering power losses. *European Journal of Operational Research*, 270(3):917–930, 2018.

[7] Matteo Fischetti. *Lezioni di Ricerca Operativa*. Libreria Progetto, Padova, 1995.

[8] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11–15, Pasadena, 2008.

[9] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[10] Ibm. *IBM ILOG CPLEX Optimization Studio CPLEX User's Manual*, 2011.

[11] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Paramount, 2009.