Heuristic algorithms for the Wind Farm Cable Routing problem

Cazzaro, Davide; Fischetti, Martina; Fischetti, Matteo

Published in:
Applied Energy

Link to article, DOI:
10.1016/j.apenergy.2020.115617

Publication date:
2020

Document Version
Peer reviewed version

Heuristic Algorithms for the Wind Farm Cable Routing Problem

Davide Cazzaro[1] Martina Fischetti[1,2]
Matteo Fischetti[3]

[1] Vattenfall BA Wind, Jupitervej 6, 6000 Kolding, Denmark

[2] DTU, Produktionstorvet 424, 2800 Kongens Lyngby, Denmark

[3] DEI, Università di Padova, via Gradenigo 6/A, 35100 Padova, Italy

e-mail: {*davide.cazzaro, martina.fischetti* }*@vattefall.com; matteo.fischetti@unipd.it*

June 23, 2019

Abstract

The Wind Farm Cable Routing problem plays a key role in offshore wind farm design. Given the positions of turbines and substation in a wind farm, and a set of electrical cables, the task is to minimize the cost of the connecting cables that are needed to transfer the power produced by the turbines to the substation, with the complicating constraint that crossing cables are not allowed. In the present paper we introduce, implement and test five different metaheuristic schemes for this problem: Simulated Annealing, Tabu Search, Variable Neighborhood Search, Ant Colony Optimization, and Genetic Algorithm. We also propose a new construction heuristic, called Sweep, that typically finds an initial high-quality solution in a very short computing time. We compare the performance of our heuristics on two datasets: one contains instances from the literature and is used to tune our codes, while the second is a large new set of realistic instances (that we make publicly available) used as a test set. According to our experiments, Variable Neighborhood Search obtains the best overall performance. Tabu Search is our second best heuristic, while Genetic Algorithm and Simulated Annealing perform worse but are often able to slightly improve the Sweep initial solution. Ant Colony Optimization, instead, struggles even to reach feasible solutions.

Keywords: (O) Wind Farm Optimization – Cable Routing Problem – Metaheuristics – Computational Analysis.

1 Introduction

Wind energy plays a key role among renewable technologies in reducing the impact on the environment caused by ever increasing power consumption. In 2017, wind provided 5% of the world electricity demand, with an increase of 52.3GW of wind power capacity added from the previous year [50]. Therefore, solving effectively the optimization problems arising wind farm design is more and more relevant for companies in this sector.

Designing an efficient wind farm leads to many challenging optimization problems, whose solution is not only of academic interest but can also bring important economical advantages to all companies involved. This is witnessed by the impressive savings reported in 2019 by the energy company Vattenfall, one of the six finalists of the prestigious Franz Edelman Award [16].

In the present work we address the *Wind Farm Cable Routing Problem* (WFCRP): given an offshore wind farm with the positions of turbines and substation, and a cable set, we want to optimally place the cables between turbines so that all the produced energy eventually reaches the substation and then can be transported to the onshore grid through a single export cable. It can be estimated from [41] that, for offshore wind farms, the electrical infrastructure costs amount to 4-5% of the total CAPEX cost, so it is important to design solutions as close as possible to the optimum.

In our scenario the positions of the turbines and of the substation have already been identified; optimizing the turbine position is on fact another important optimization problem in this application field that has been studied, among others, in [19, 38, 46, 4, 14, 15].

WFCRP has been proved to be NP-hard [26] in theory, and also very hard to solve to proven optimality in practice, even using advanced Mixed-Integer Linear Programming (MILP) models and the best solvers on the market. In practical applications involving preliminary what-if analyses, however, one is mainly interested in finding reasonably good solutions in very short computing times, a setting that motivated our interest for new heuristic techniques (not using MILP technology) to solve it.

The paper is organized as follows. Section 2 gives a precise definition of WFCRP and of the corresponding constraints, namely: guaranteeing that the solution defines a tree; ensuring that only one cable exits each turbine; using for each link a cable type with enough capacity to support its power flow; not exceeding a given maximum number of connections to the substation; avoiding cable crossings that would imply large extra costs. In this section we also examine the existing literature on the problem, which tries to solve it both with MILP models and with ad-hoc heuristics.

Section 3 describes our experimental design, and in particular the use of two

different sets of instances for training (i.e., code tuning) and testing (i.e., benchmarking) the developed codes. Our training set is composed of 24 real-world instances of real wind farms taken from the literature [26]. The test set involves instead 220 medium-to-large size new instances which are available, on request, from the authors.

Section 4 describes some basic tools used by the heuristics. Our six new heuristics are introduced in Sections 5 to Sections 10. The first one, `Sweep`, is a constructive heuristic which is able to compute very good solutions in just a few seconds of computing time, hence it is used as a warm start by the other techniques. The other five heuristics are based on metaheuristic strategies: Simulated Annealing, Tabu Search, Variable Neighborhood Search, Ant Colony Optimization and Genetic Algorithm. All heuristics benefit in efficiency by considering a clever parametric cost evaluation when computing the solution costs. For each (meta)heuristic we report the implementation details, the variants and the parameters considered during their development.

In Section 11 we analyze the performance of the heuristics on our two sets of instances. Finally Section 12 summarizes the work done and the results obtained, and provides some suggestions for future researches.

2 The Wind Farm Cable Routing Problem

WFCRP can be outlined as follows; see [26] for fuller details. Assuming the turbine positions have already been selected for a specific farm, the task is to choose which cables to place between the turbines to transfer all the produced electric power to a given collecting point (the substation), that eventually is connected to the power grid on the coast. Different cable types are available, each characterized by a different unit cost and electrical capacity.

We limit the problem of interest to wind farms with only one substation, as it is in the majority of real cases, and only consider offshore wind farms that do not need to take terrain morphology or obstacles into account.

2.1 Solution structure

On input, we are given the following information:

- the number $n+1$ of points in the site: point 0 corresponds to the substation, while points $1, \ldots, n$ correspond to the turbines;
- the fixed 2D-coordinates (x_i, y_i) of each point in the site, $i = 0, \ldots, n$;
- the power production P_i of each turbine $i = 1, \ldots, n$;

- a set of K cable types, the k-th of which having a capacity $capacity(k)$ and unit cost c_k, $k = 1, \ldots, K$;
- an upper bound C on the number of cables that can be physically connected to the substation.

Typically, all turbines are of the same type and hence produce the same power, so power productions and cable capacities are normalized, i.e., we have $P_i = 1$ for $i = 1, \ldots, n$ (while the substation has no power production).

Points in the site can be visualized as nodes of a complete and loop-free graph $G = (V, E)$, whose edges correspond to the potential cable connections. On output, we have to find a tree on G whose arcs correspond to the built cable connections. Arcs in the tree are directed from the turbines to the substation, according to the electrical flow, so they define an anti-arborescence rooted to node 0 where the out-degree of each non-root node is 1. Figure 1 plots an example of a feasible solution.

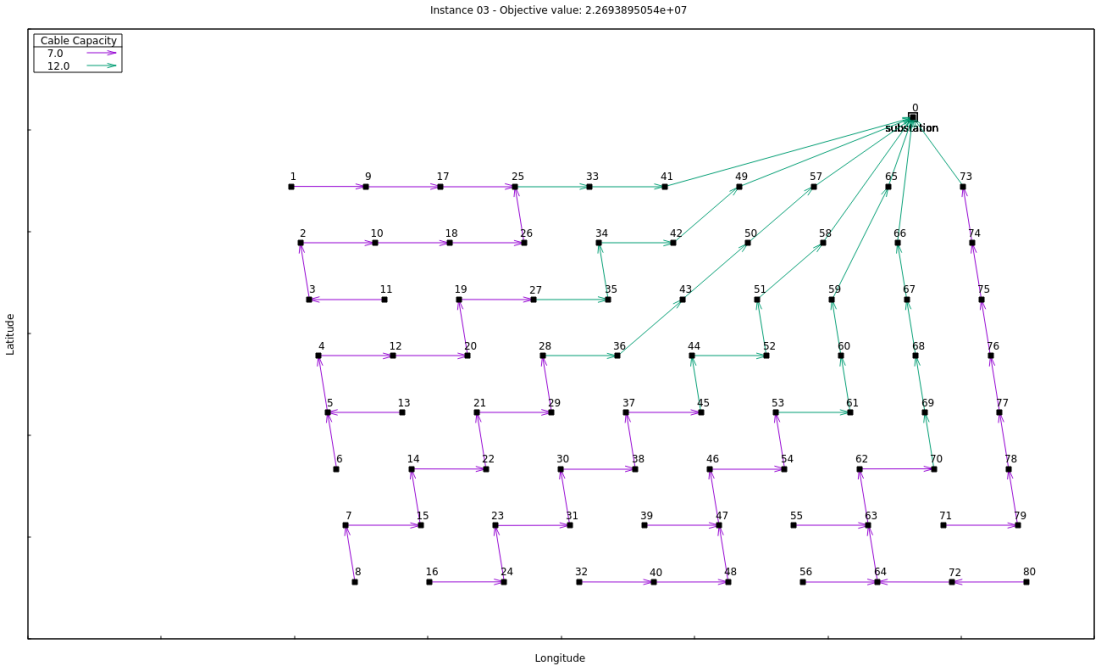

Figure 1: Example of a solution for a WFCRP instance with $n = 80$ turbines. The substation is node 0, all other nodes are turbines with power production $P_i = 1$. Cable capacities are normalized: cable type 1 can support up to 7 turbines, while cable type 2 is more expensive but can support up to 12 turbines. At most $C = 6$ arcs can enter the substation, and the built cables are pairwise noncrossing.

2.2 Penalized cost function

Given any feasible solution $G_T = (V, T)$, say, one can easily compute the total amount of flow f_{ij} on each arc $(i, j) \in T$, and choose the less-expensive cable type $k(i, j) \in \{1, \ldots, K\}$ that supports such a flow f_{ij}. The overall cost of the tree is therefore easily computed as

$$cost(T) := \sum_{(i,j)\in T} c_{k(i,j)} \cdot dist(i, j),$$

where $dist(i, j)$ gives the Euclidean distance between points i and j in the plane. For the heuristics, however, the cost function must be extended to include penalties for possible sources of infeasibility such as crossings, extra-flow on some arcs, extra cables to the substation, etc.

The heuristic techniques we developed use a *successor* data structure to represent a solution: it memorizes for each node $v \in V$ which node the exiting arc (which is only one for each node) is connected to. The penalized cost function is then computed according to the following steps:

1. *Compute arc flows*: Calculate the flow on each arc, using a modified Depth First Search algorithm [48] that sums all the produced power from leaf nodes to the root node (the substation).

2. *Select cable types*: Based on the flows computed at the previous step, a cable type is assigned to each arc by selecting the cheapest cable that can support the power flowing through it; in case no such cable exists, we select the maximum-capacity cable available and mark the arc as an *overflow arc*.

3. *Define cable costs*: We sum the unit costs of each selected cable multiplied by its length; for overflow arcs, and add a penalty of M_1 (say) to the total cost, multiplied by the amount of extra flow.

4. *Connections to the substation*: We count how many arcs are connected to the substation: if this number exceed C, we add to the total cost a penalty of M_2 (say) for each extra connection.

5. *Crossings*: We count how many arcs cross any other arc in the solution: for each crossing, a penalty of M_3 (say) is added to the solution cost.

6. *Proper tree*: Finally, to be sure that the solution has a meaningful structure, we penalize by an exceedingly large value M_4 (say) any solution containing self-loops, cycles, disconnected components, or a number of arcs different from n.

In our implementation we used the following very-large penalties for infeasibilities: $M_1 = M_2 = M_3 = 10^9$, and $M_4 = 10^{10}$. As to the crossing penalty M_3, we observe that for a company it could be interesting to know if the extra-cost to allow them would pay off. To this end, in our heuristic framework one can simply allow them and set penalty M_3 to the actual cost of building a crossing—while the same would be prohibitive in a MILP model as it would require a huge number of additional variables.

2.3 Related work

WFCRP has been studied by many authors, using both exact and heuristic techniques.

The work of [40] considers a hybrid genetic and immune algorithm and tests it on a offshore wind farm. Genetic algorithms have been studied also in [51] and [31]: the first reference considers power losses, wind power production, initial investment and maintenance costs in their optimization; the second one is modified to take into account different cable cross sections in the design of a radial array.

A heuristic technique has been studied in [8], which uses a multi-level clustering for the cable routing.WFCRP is instead modeled as a planar open vehicle routing problem in [1]; this formulation imposes the constraint that only one cable can enter a turbine.

In [35] a heuristic based on minimum spanning trees is developed and enhanced with an adaptive particle swarm optimization; this heuristic is then compared, on three offshore wind farm scenarios, to deterministic methods based on minimum spanning trees and on a dynamic variant of it. The work [52] presents an algorithm of self-tracking minimum spanning trees that works with a fuzzy C-means clustering.

In [2] a divide-and-conquer solution approach is proposed, although it is tested only on small wind farms ($n < 11$). Turbine placement and cable routing are studied in [9], where a MILP model for each of these two problems is proposed.

The paper [26] proves the NP-hardness of the problem even in some relevant special cases, and presents a MILP model along with exact and so-called matheuristic [12] solution approaches. The proposed model can take power losses and obstacles into account, and exploits so-called Steiner nodes to allow for bended cables. A computational analysis is reported, and a set of 24 real-world instances is made publicly available for benchmarking. Several extensions of the basic MILP model have been proposed in [13, 20, 21, 25, 23, 24, 22, 11].

The work [3] also considers power losses in cables and develops MILP models for the problem. In [34] the problem is studied for onshore cases. The authors develop a Mixed Integer Quadratic Programming model, which is then simplified to a MILP model to be solved by standard software. They also consider two cable

types: underground and on the ground. For the first cable type parallel structures are preferred since they are easier to build, while existing roads are favored for ground cables. In [37], instead, a MILP model is developed for wind farms, taking obstacles into consideration. MILP solutions are compared in [45] with a k-means technique for obstacle avoidance in cable routing.

3 Experimental design

Heuristic methods typically involve a number of parameters affecting (often in an unpredictable way) the quality of the final solution found. To overcome the risk of overtuning, we decided to use two separate sets of instances for tuning and for testing.

3.1 Training Set

Our training set is composed of 24 instances from [26]. The data refers to five real wind farms located in the United Kingdom, Germany and Denmark, and are summarized in Table 1.

Name	Site	Turbine Type	# of Turbines	C
wf01	Horns Rev 1	Vestas 80-2MW	80	10
wf02	Kentish Flats	Vestas 90-3MW	30	∞
wf03	Ormonde	Senvion 5MW	30	4
wf04	DanTysk	Siemens 3.6MW	80	10
wf05	Thanet	Vestas 90-3MW	100	10

Table 1: The five real-world wind farms used for training [26]

.

The instances that correspond to the second and third wind farms are the easiest ones, having only 30 turbines, while the most difficult instances are the last four ones, which correspond to a wind farm with 100 turbines. As a matter of fact, many of the harder instances do not reach a feasible solution using an exact solver for the MILP model, and require sophisticated matheuristic techniques [12] to reach good (although possibly suboptimal) solutions.

3.2 Test Set

Our test set has been created on real-world data with the aim of considering realistic case scenarios. In particular, the combinations of the following items have been explored to create our wind farm layouts:

- n: we considered 3 different values for n, namely, 50, 80 and 120.
- s: we considered 5 different areas, which are based on real wind farm sites.
- t: we considered 4 different turbine models (namely: t01, t02, t03 and t04) with different rotor diameters and power productions (respectively: 8MW, 7MW, 8.4MW, and 8MW); as in most real cases, each wind farm is made of turbines of the same type.
- w: we considered 5 different real wind statistics for each layout.

The instance names are composed by the combination of the parameters above with their numeric value; for example, instance *n50_s01_t02_w03* indicates a layout of 50 turbines (n50) for the first site (s01), with turbines of the second type (t02) in the third scenario of winds (w03).

The combination of these parameters leads to 300 different cases. For each case, an optimized turbine layout has been computed with a 1-hour time limit, using the MILP-based matheuristic in [17, 19] using the Proximity Search paradigm [18]. A post-processing phase was required to exclude some of the obtained layouts, since not all of them admitted a feasible position for all the turbines (especially for the largest values of n). This was due to the fact that the layout optimization must satisfy some constraints such as not placing the turbines too close one to each other (and in some cases the available area was not enough to accommodate all turbines). For this reason 80 layouts had to be eliminated from the test set, leaving 220 valid layouts with the following composition: 100 layouts with 50 turbines, 80 layouts with 80 turbines, and 40 layouts with 120 turbines.

Cost per meter	7MW turbines	8MW turbines	8.4MW turbines
430	6	5	5
480	8	7	7
610	14	12	11

Table 2: The cable set used for the test instances. We report a realistic unit cost for the three cable types, along with the number of turbines they can support for different turbine types. Unit costs include installation costs.

The cable set used for the test instances is given in Table 2. For each layout we considered three possible positions (x_0, y_0) for the substation, as shown in Figure 2. The default position we consider for the substation is in the middle-top of the turbine layout, as this resembles what happens in some real cases of the training set. The other two options are on the middle-left, and on the geometric center (barycenter) of the turbines.

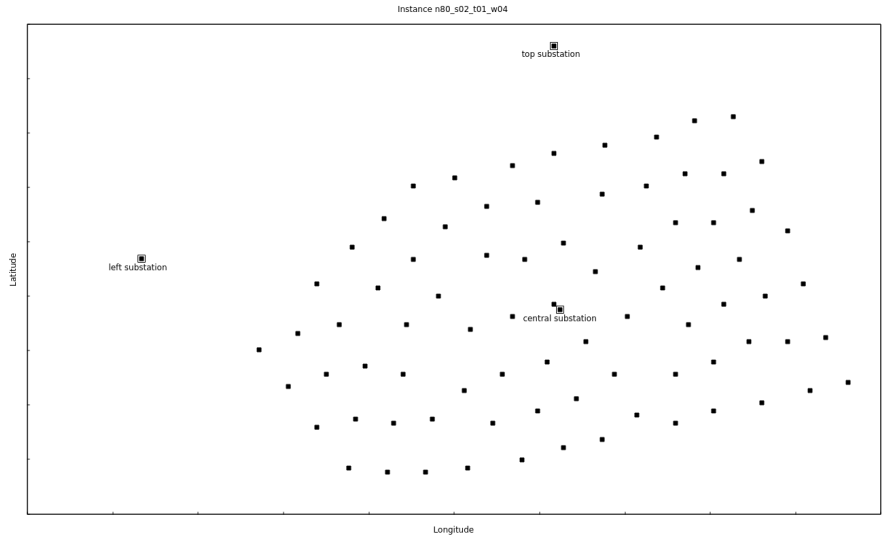

Figure 2: An example of the three substation positions, for test instance n80_s02_01_w04.

The last parameter to consider for the test instances is C, i.e., the maximum number of cable connections to the substation. According to our computational experience, the value of C together with the maximum cable capacity and the number of turbines gives a rough indication of the difficulty to find a feasible solution for that instance. As we are interested in generating challenging instances for the heuristics, for each instance we first computed the minimum feasible value for C as $C_{min} = \lceil \sum_{i=1}^{n} P_i / \max\{capacity(k) : k = 1, \ldots, K\} \rceil$, and then considered instances with $C = C_{min}, C_{min} + 1, \ldots$ where the constraint becomes looser and looser. In order to have difficult instances, but also not to be too close to the extreme case $C = C_{min}$, we choose $C = C_{min} + 1$ as the default input value for our tests.

4 Basic heuristic tools

We next describe the two algorithms we used to obtain an initial (possibly infeasible) solution, and report two of the building blocks used by the other heuristics: the *1-opt move*, which is the basic step used to improve a solution, and the *Parametric Cost Evaluation*, which greatly boosts the computationally intensive phase of computing the penalized cost of a solution.

4.1 Prim-Dijkstra

As a basic tool we use the Prim-Dijkstra algorithm [5] to compute an undirected Minimum Spanning Tree, whose edges are oriented towards the substation in a post-processing step. The solution found was often infeasible feasible because the few arcs connected to the substation had to support a large number of turbines each, but none of the available cables had enough capacity. Nevertheless, it is interesting to compute the Minimum Spanning Tree as it would be the optimal solution for our problem if we had only one type of cables and no constraints other than connectivity. In addition, Prim-Dijkstra algorithm can be used to provide an initial (likely infeasible) solution for other heuristic techniques. Moreover, by considering the cheapest cable for each edge in the Minimum Spanning Tree and ignoring if the power exceeds the cable capacity, the Prim-Dijkstra solution can provide a rough lower bound on the cost of the optimal solution. Such an estimate is fast to compute, having $\mathcal{O}(n^2)$ time complexity with a proper implementation, and turns out to be useful, e.g., in our Simulated Annealing heuristic.

4.2 GRASP

The Greedy Randomized Adaptive Search Procedure (GRASP) [10] is a meta-heuristic used for combinatorial problems and leverages on a multistart scheme. In our implementation we modify the Prim-Dijkstra Minimum Spanning Tree algorithm as follows: at each step, instead of choosing the minimum cost edge, we create a Restricted Candidate List (RCL) with the first k best choices. From this list we select with probability p the best choice, that is the same edge that the Prim-Dijkstra algorithm would have chosen and that corresponds to the first candidate in the RCL list. With probability $1 - p$, instead, we pick at random one of the other elements in the RCL list, so we add to the solution an edge that can be the second-best option, the third-best one, etc.

The two parameters p and k must be tuned to have enough variance in the RCL list but not too much, otherwise it would generate just random solutions. While the GRASP solutions are in general not feasible, we exploit their variety of solutions to generate initial solutions in the Genetic Algorithm, see Section 10.

4.3 1-opt move

To pass from a solution to a close one in the solution space, in many of our heuristics we exploit a 1-opt move. For our specific problem this move consists in replacing just one arcs in the solution: we select at random a turbine node in the tree, say i. Because of the constraints on our problem, node i will only have one leaving arc (i, j) (say). Therefore we remove arc (i, j) and connect i to another

node of the tree, excluding the same node i and all the nodes of its connected component (computed by moving backwards from i to its leaves) to avoid cycles.

4.4 Parametric cost evaluation

When computing the penalized solution cost, we have to compute each time the power that flows through each arc and the minimum-cost cable-type needed for each arc, and to count the number of arcs entering the substation and the number of edge crossings. A better way is to maintain some auxiliary information about the current solution and to update it incrementally by exploiting the fact that each 1-opt move only changes one of its arcs. We are thus able to reduce the complexity of a single 1-opt move from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$, which greatly boosts the performance of our algorithms.

5 Sweep

We next propose a new constructive heuristic, called `Sweep`, that plays a prominent role in our study. Our key observation is that, in many cases, the main constraints are given by the maximum number of connections to the substation and by the maximum capacity of cables. Thus, a main difficulty for the heuristics is to partition the turbines in a balanced way, so that the turbines in each group correspond to a subtree of the substation and thus have a total production that can be supported by a single cable. This makes our problem similar to the Bin Packing problem [6], since we have to decide which turbines to *pack* together in which group. Although NP-hard in the general case, the Bin Packing problem is typically not difficult to solve heuristically, and becomes trivial in case $P_i = 1$ for all items—as it is often the case in the WFCRP context.

Our heuristic has three parameters:

- *starting turbine* from which the algorithm starts;
- *direction* when scanning the turbines (clockwise or anti-clockwise);
- *turbines per group*, abbreviated to TPG.

First we order the turbines by the angle defined with respect to the substation. Then we pick a node as the starting turbine, and *sweep* the other turbines clockwise or anti-clockwise according to the chosen direction. The actual number of turbines that we will put in each a group, TPG, is an integer between $L_1 = n \,/\, C$ and $L_2 = max\_cable\_capacity \;/\; avg\_power$, where $max\_cable\_capacity$ is the capacity of the biggest available cable and avg_power is the average production of a turbine (=1 in the cases of interest).

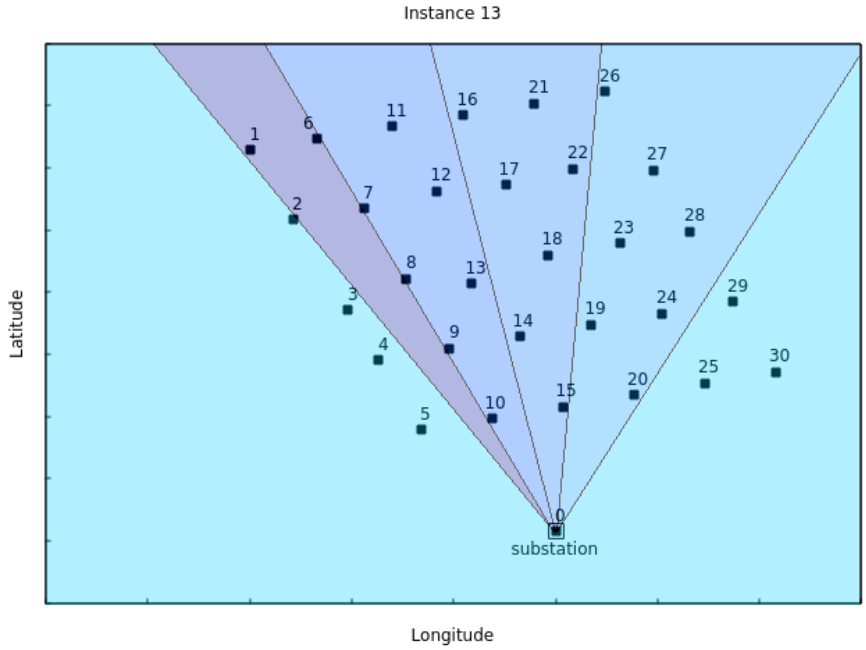

Figure 3: An example of the turbine partition of the `Sweep` heuristic: starting from turbine 2, in anti-clockwise direction, with 7 turbines per group. It forms five groups of turbines (with different colors in the figure): four groups have 7 turbines each, the last one is composed of just two turbines (1 and 6).

Having decided TPG, we add turbines to a group following the defined turbine order, until we reach the required number TPG (thus, only the last group can contain less than TPG turbines), as shown in Figure 3.

After defining the turbine groups, we set the edge costs to infinity for each edge among turbines of different groups, except for the edges connected to the substation, and finally compute the Minimum Spanning Tree on these modified edge costs using the Prim-Dijkstra algorithm, thus choosing the best edges inside each group of turbines and towards the substation. In a final post-processing step, the selected edges are oriented such that all the energy flows eventually to the substation, each edge is assigned the less expensive cable with enough capacity to support its flow, and the penalized solution cost is computed.

5.1 Tuning

We implemented `Sweep` within a multistart scheme. As there are only $\mathcal{O}(n^2)$ combinations of the three parameters (the starting turbine, the direction and the number of turbines per group) we can afford to try each of them. In practice

this is extremely fast, as we are able to evaluate all the combinations in a very short computing time—at most a couple of seconds, on a PC, even for the largest instances of our testbed.

The `Sweep` heuristic has proven to be very effective in finding good feasible solutions in very short time, even for the most difficult instances of our training set. For this reason it has been included as a warm start for many of the other heuristics we implemented. This allows the other heuristic techniques to start from an already feasible (and typically quite good) solution—this is a nontrivial advantage, as for the hardest instances many heuristics could not even reach feasibility at the time limit when starting from a Prim-Dijkstra or GRASP solution.

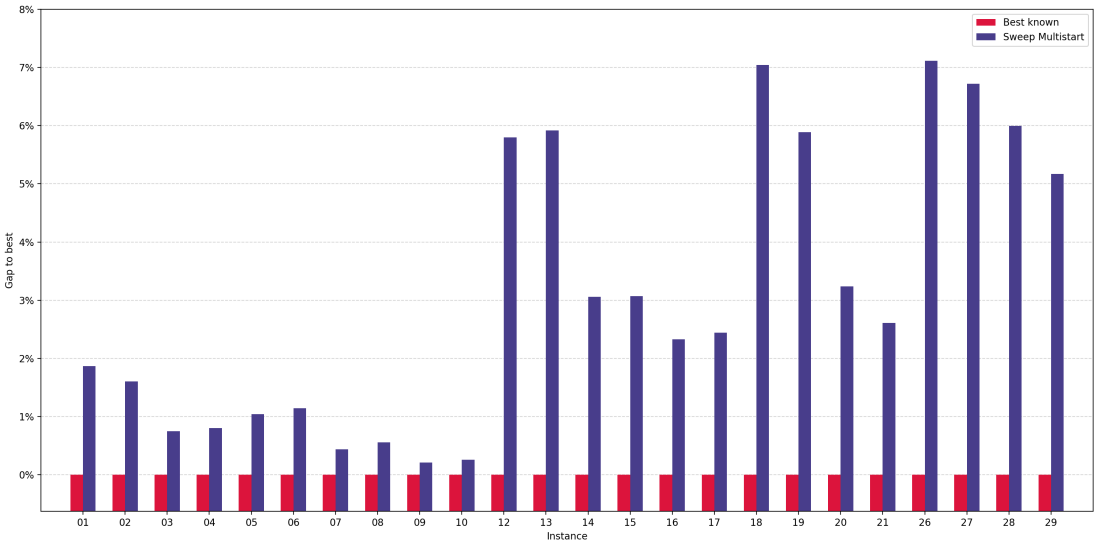

Figure 4: Gap obtained by `Sweep` (multistart) with respect to the best known solutions [26] on the training instances (the lower the better). The heuristic, in less than one second of computing time, reaches quite good solution costs, all within 8% from the best known ones.

Figure 4 compares the gap between the `Sweep` solutions with the results of the MILP-based matheuristic used in [26]. It is interesting to observe that, for all instances, the `Sweep` cost is within 8% from the optimal or best-known solution. Also note that the results from [26] uses one hour of computation time, while the `Sweep` best solution is computed in less than one second for each training instance.

6 Simulated Annealing

Simulated Annealing (SA) [36] is a heuristic method that takes inspiration from physical processes, in particular from how cycles of controlled heating and cooling lead the molecules of a material to reach a crystal or electronic configuration of minimal energy. For example this process is used in metallurgy to gradually remove imperfections in the metal in order to make it stronger, which can be seen as a minimum energy problem. The phase of *reheating* the problem represents a crucial point for the heuristic, allowing the technique to exit from points of local minimum.

In parallel with the physical world, the main parameter for the Simulated Annealing strategy is called *temperature* (T) and it is updated at each step as $T_{new} = \alpha\, T_{old}$, where $\alpha \in (0, 1)$ is a parameter that controls cooling speed. The other important feature of SA is the probability of accepting a *move*, which favors the procedure of passing from a solution to another one. We want to always accept an improvement in term of *energy* of the solution, as the cost is called in the SA jargon, and also to allow for a worsening move with good probability only when the temperature is high. Additionally, the acceptance probability must depend on the magnitude of the difference between the energy of the candidate move and the energy of the current solution, namely, $\Delta E = E_{candidate} - E_{current}$ (positive if the candidate solution is worse that the current one) The acceptance probability p is typically computed as follows [36]:

$$p = \begin{cases} e^{-\frac{k\Delta E}{T}} & \text{if} \quad \Delta E > 0 \\ 1 & \text{if} \quad \Delta E \leq 0 \end{cases} \tag{1}$$

where T is the current temperature, ΔE is the difference of energy, and k is a normalization factor.

Finally, we should choose a T_{min} threshold that determines the stopping condition for the heuristic but, since our techniques are all tested with a time limit, we modified this part of the algorithm so that it runs until the available time expires.

Figure 5 reports the variation of the energy (i.e., solution cost) for the first few thousand iterations. We can see how, in the initial high-temperature phase, the heuristic explores very bad solutions, and how the energy tends to improve over time. Furthermore we can see when a local minimum is reached and how the heuristic, with a series of iterations that are initially high in cost, is able to escape from it and to find another one that has a lower energy. With the progressive cooling of the temperature, the accepted solutions have less energy difference and bring us closer to global minimum.

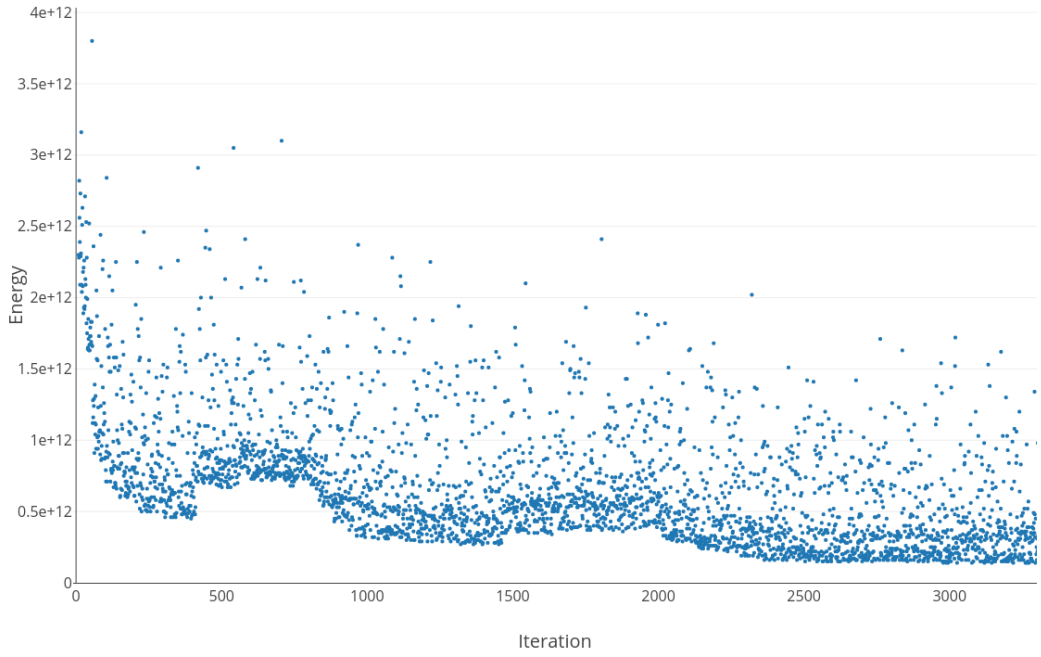

Figure 5: The energy of the explored solutions by the Simulated Annealing heuristic for the first few thousand iterations. The overall energy (solution cost) gradually diminishes; at around iteration 500 the heuristic escapes a local minimum with a series of worsening moves that eventually lead to a lower cost. The same happens around iteration 1500. Note that, in the first iterations, the technique considers very bad solutions: this is due to the high initial temperature that is then reduced over time.

6.1 Tuning

Since SA needs an initial solution, we decided to use the `Sweep` multistart heuristic to generate it, allowing the algorithm to start from an already feasible solution.

In our implementation, we started with an initial temperature $T_0 = 1.0$ and tried different values for $\alpha = 0.9, 0.99, 0.999, 0.9999, 0.99999, 0.999999$ as shown in Figure 6. The performance was similar for many of these values, meaning that the descending function was not too fast nor too slow. We selected the value $\alpha = 0.99999$ as it produced slightly better results for the harder training instances.

As an additional improvement [39], we kept the temperature fixed for a certain number of iterations, that we set to twice the number of turbines in our problem.

Furthermore, since in our case the ΔE can be very large (typically ranging from

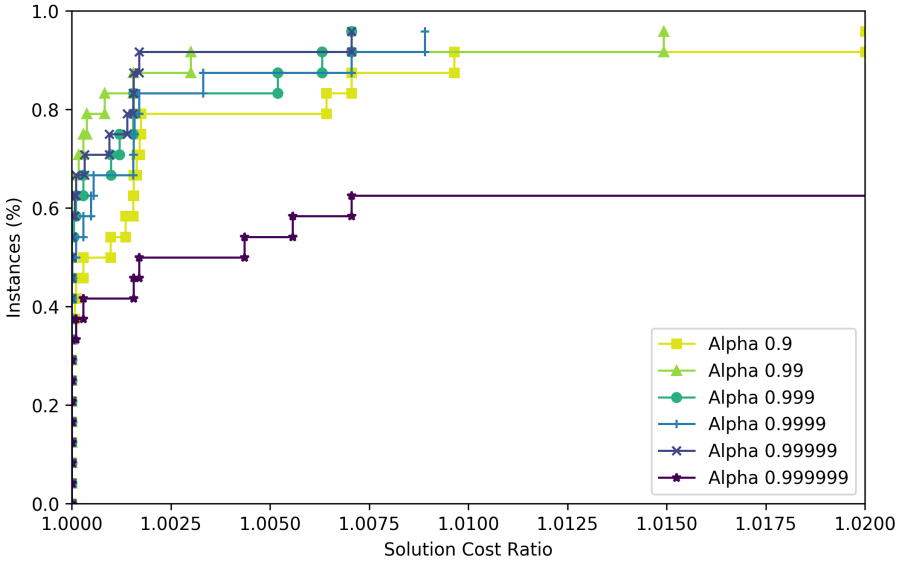

Figure 6: Performance profile comparison of Simulated Annealing on the training instances with six values for the α parameter. Many profile curves are close together, with value $\alpha = 0.99999$ being slightly better on harder instances.

10^6 to 10^8), we set $k = \frac{1}{LB}$ where LB is the non-penalized cost of the (possibly infeasible) solution computed by the Prim-Dijkstra algorithm.

Another important point is how to generate a candidate move in the Simulated Annealing heuristic. We chose to exploit the 1-opt move where a single arc is changed from the current solution to the alternative one, hence ΔE can be computed very efficiently through the parametric cost evaluation of Subsection 4.4.

7 Tabu Search

Tabu Search (TS) [28] is a metaheuristic that tries to escape from the local optima reached by a local search algorithm. The main idea is to use a memory structure called *Tabulist* [30], which registers which moves the algorithm is not allowed to repeat. This allows the heuristic to reach new solutions while avoiding to return to a previous local optimum. This technique has been used for many problems, often leading to very good results [29].

The Tabulist does not forbid a particular move or condition forever, but for example for a number of iterations since it occurred. This memory mechanism is based on a parameter called *Tenure* and ensures that, while we do not want to repeat the same moves too soon, we allow to repeat them after a while, to

hopefully reach a new local optimum.

7.1 Tuning

We initialized the Tabu Search with a `Sweep` multistart solution. As in the other heuristics we implemented, we enforced a time limit as a stopping condition.

For the candidate list of moves, we considered all $\mathcal{O}(n^2)$ 1-opt moves and took the best one. As the 1-opt move changes one arc (i, j) of the solution with another arc (i, k), say, the Tabulist can be made in different ways. We decided to consider three options:

- *Node From*: we add node i to the Tabulist ;
- *Node To*: the Tabulist registers which was the final node k;
- *Arcs*: the Tabulist considers the new arc (i, k) itself, i.e., the newly chosen arc cannot be removed too soon.

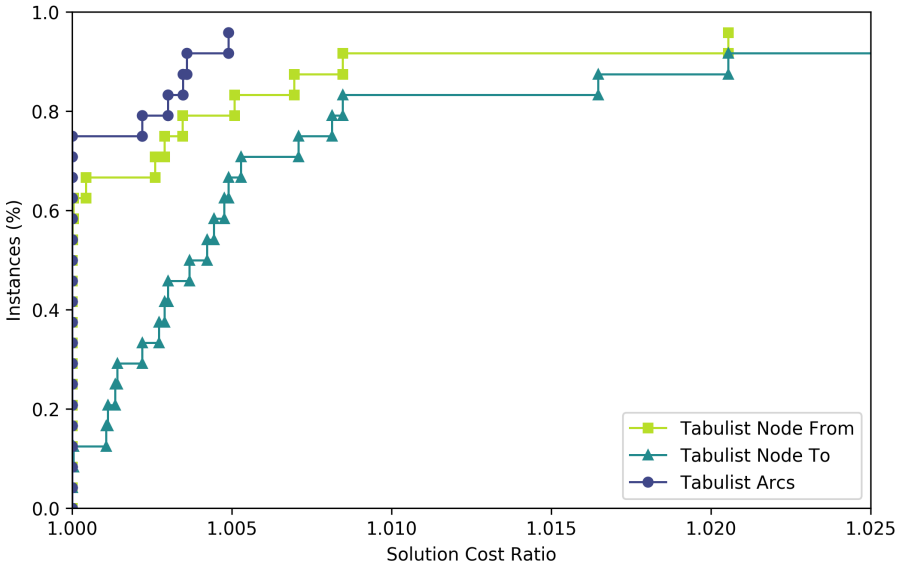

Figure 7: Performance profile comparison of Tabu Search on the training instances with three different Tabulist choices: Node To, Node From and Arcs (the last being the most effective one).

In Figure 7 we give a performance profile comparing the three options on the training set. Choosing to add the arcs to the Tabulist is the most effective technique, while acting on nodes seems to be too restrictive. Arcs and Node From

had a very similar performance in 60% of the training instances (namely, the easiest ones), while the difference became relevant for the hardest ones.

Finally, we used two strategies to favor diversification [27]: after a certain number of iterations from the last incumbent improvement, 50% of the times we either change at random the Tenure parameter or reset the Tabulist. The new Tenure value is chosen at random but we ensure that it is not lower than half the number of turbines.

8 Variable Neighborhood Search

Variable Neighborhood Search (VNS) [42] is a metaheuristic that searches for a global optimum by exploring increasingly distant neighborhoods of the current incumbent solution. It consists of three main steps [33], that are iterated until a time limit is reached. We now analyze in details its components:

- *Shaking*: generate a solution y' at random from the k-th neighborhood of the incumbent solution y. For our problem we make k consecutive random 1-opt moves. As starting solution for the first iteration we get the solution from `Sweep` multistart heuristic.

- *Local Search*: starting from y' apply a local search method. We do so by finding the *best* 1-opt move (among the $\mathcal{O}(n^2)$ possibilities) until the solution cost stops improving. (We also tested the *first improving move* policy, interrupting the 1-opt search as soon as an improvement is found instead of considering all possible moves, but this option led to a worse performance on the training instances.) Let y'' denote the solution found at the end of this local-search phase.

- *Move or Not*: if the new solution y'' has a better cost than the incumbent y, we update $y = y''$ and the k neighborhood counter is reset to 1. If the solution is not improved, we increment the neighborhood by setting $k = k + 1$, until a maximum neighborhood size of K_{max} is reached (afterwards, threshold k is reset back to 1).

8.1 Tuning

One of the advantages of the VNS heuristic over other heuristics is that it has basically just one parameter, K_{max}, to tune. In a preliminary tuning phase, however, we also explored some VNS variants. When generating the new solution y', we tried starting from the current solution instead of restarting from incumbent solution, but this worsened the performance probably because we were drifting too

much away from where the good solutions are located. Two mixed approaches, namely restarting sometimes from the incumbent and other times from current solution, or restarting from the incumbent solution only when parameter k is reset, led to comparable or slightly worse results than the original option, so in the end we decided to always restart from the incumbent solution.

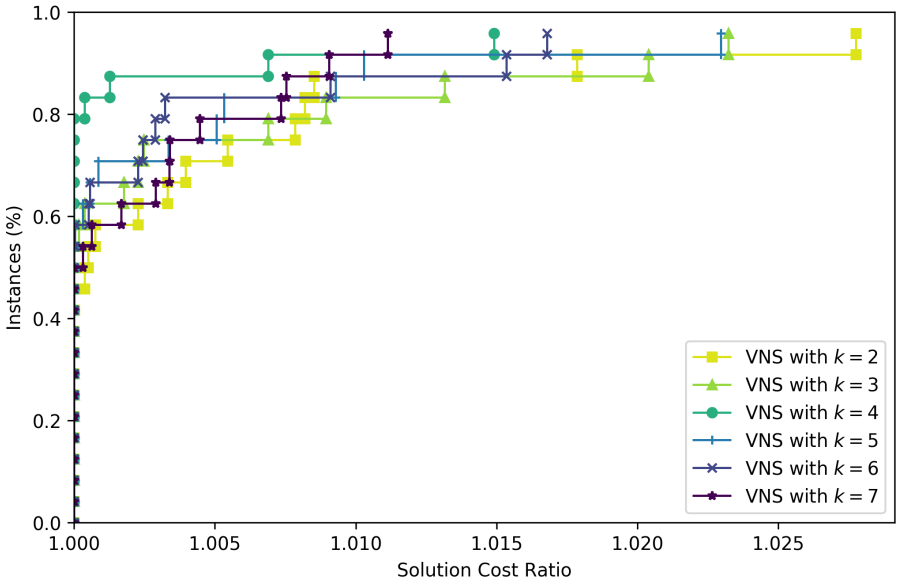

Figure 8: Performance profile comparison on training instances varying parameter k for the VNS heuristic. It can be seen that $K_{max} = 4$ has better performance than the other values.

As to parameter K_{max}, in the shaking phase we tried $K_{max} \in \{2, 3, 4, 5, 6, 7\}$. The results on the training instances are reported in Figure 8, where it can be seen that using a value of $K_{max} = 4$ (a good compromise between differentiating the neighborhood to explore and shifting too far from good solutions) led to the best results.

9 Ant Colony Optimization

Ant Colony Optimization (ACO) [7] is a metaheuristic that mimics the behavior of ants when finding their food. In nature each ant wanders around randomly, looking for food and leaving a trace of pheromones along its path. When a source of food is found, the ant goes back to the anthill, thus reinforcing the amount of pheromones on that particular path. The other ants detect them and then are more likely to follow paths with higher concentration of pheromones. In this way the paths that lead from the anthill to a food source are reinforced. Also, because

of the initial random fluctuation, the shorter path are privileged, since ants return sooner to the anthill and the pheromones have less time to fade compared to the amount on longer paths. ACO takes inspiration from this mechanism and has been used for example to solve the Traveling Salesman Problem [7]. For the WFCRP, the work [43] compares various implementation strategies. Their metaheuristic follows the steps reported in Algorithm 1.

Algorithm 1 Ant Colony Optimization for WFCR problem

```
start_time ← now()
pheromones ← init_pheromone_values()
while now() − start_time < time_limit do
  y_current ← find_path(pheromones)
  pheromones ← update_pheromone_values(pheromones, y_current)
  if penalized_cost(y_new) < penalized_cost(y_incumbent) then
    y_incumbent ← y_new
  end if
end while
```

First we have the find_path function, that retrieves a solution for the problem based on the pheromones. Then the update_pheromone_values function simulates the pheromone concentration decay over time, while reinforcing it in the edges chosen for a solution.

The most crucial part of this algorithm is in how the find_path function works. In [43] various techniques are considered to adapt this part to our problem. The best option they identify is called the Kruskal approach, that prescribes to select a new edge at a time, building a new solution from a neighborhood that contains all edges that do not create a cycle with the already built partial solution.

The probability of choosing one edge from this neighborhood is given by a probability computed with respect to the pheromone amounts. The probability used in [43] reads:

$$P_e \;=\; \eta(e)\,\tau(e) \sum_{\substack{x=\{c,v\}\in E \\ v\notin V(y_{\text{partial}})}} \eta(x)\tau(x)$$

where P_e is the probability of choosing edge e, function $\eta : E \to \mathbb{R}$ is $\eta(e) = \text{length}(e)^{-1}$, $\tau : E \to \mathbb{R}$ is the amount of pheromones on edge e, and $V(y_{\text{partial}})$ are the nodes that have at least one edge that has already been chosen in the partial solution.

After an edge has been chosen based on these probabilities, it is added to the partial solution and both the probabilities and the neighborhood are updated accordingly. We iterate this procedure until all edges have been selected.

At the start of the heuristic all edges have the same amount of pheromones, say 1, that gradually decays over time thanks to the update function, except for the edges that have been selected in a solution. The new pheromone values are computed in [43] as follows:

$$\tau'(e) = \begin{cases} \tau(e)(1-p) + M \,/\, penalized\_cost(y_{current}), & \text{if } e \in E(y_{\text{current}}) \\ \tau(e)(1-p), & \text{otherwise} \end{cases}$$

where $p \in [0, 1]$ is the *pheromone decay* parameter and $M > 0$ is a fixed parameter that gives an estimate of the optimal penalized cost.

9.1 Tuning

The authors of [43] did not penalize crossings, which is indeed an issue for our realistic instances that have many more turbines than the artificial examples they considered. In order to consider and avoid crossings, we modified the find_path function based on the Kruskal approach by excluding, in addition to the edges that would create a cycle, also those edges that would produce a crossing with the edges of the partial solution. Since this change can sometimes lead to an empty list of valid edges, especially when the partial solution has many edges, we relax this condition and allow for a crossing edge to be selected (with a penalty).

Another point is choosing a good parameter M for the pheromone update formula above, that should favor solutions close to the (unknown) optimal cost and not increase much the pheromone amount on the edges chosen by worse or even infeasible solutions. To do so we estimate a good penalized solution cost by using the `Sweep` heuristic at the beginning of the algorithm, which empirically is not too far from the optimal one, and use this value to initialize M. If a lower cost value is found by the algorithm during its iterations, we update M to match the new incumbent cost. In this way we have seen that the probability values of edges, computed on the pheromone values, have the desired behavior of favoring edges chosen by many good solutions, while being influenced not too much by infeasible solutions.

For the Kruskal algorithm used in [43] we actually tried three different variants, since some details were missing in the original paper:

- Maintaining a current node c from which we can choose an edge to reach another node. In this way, however, the solutions generated are more similar to a walk than to a proper tree, and often do not reach feasibility.
- Choosing a random node from the ones in the partial solution, and choosing a new edge from there based on the given probabilities (excluding edges

that would cause cycles or crossings). This approach works better than the previous variant, reaching feasibility in many instances, but still remains far from the optimal solutions.

- Consider directly the edges, adding each one to the partial solution based on the probabilities of all admissible edges. Since the original variant is named after Kruskal, we believe this variant is closer to the one in the original paper—as a matter of fact, this technique led to a better performance than the other two interpretations, even tough it did not always reach a feasible solution for the training instances.

As further enhancements we also reset the pheromone values of edges after the empirical value of 15,000 iterations after the last incumbent improvement. This value, however, was not crucial as it did not affect the final solution quality very much.

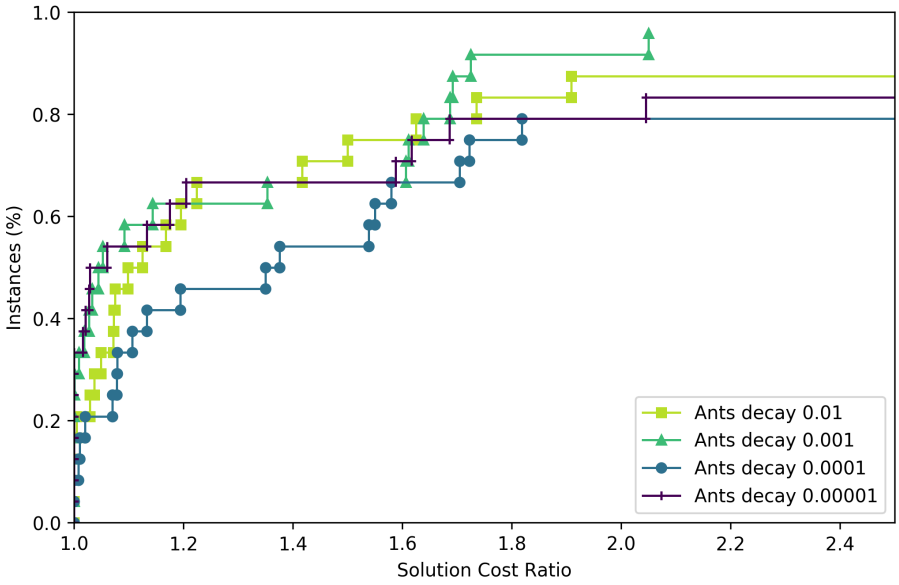

Figure 9: Performance profile comparison of different pheromones decay parameters p on training instances.

The other parameter to tune is the pheromone decay probability p, which is used to calculate edges probability when building the solutions. This parameter needs to be large enough to allow "bad" edges to be forgotten, especially if chosen in the initial stages of the algorithm when the solutions are more or less random, but not so strong to delete the previous "good" edges found in good solutions. We tested $p \in \{0.01, 0.001, 0.0001, 0.00001\}$ and we report in Figure 9 the corresponding performance profile on the training instances. The profile curves are close one

to each other; we chose $p = 0.001$ since it led to a slightly better performance on the hard instances.

10 Genetic Algorithm

Inspired by the natural selection process, the Genetic Algorithm (GA) [49] is a metaheuristic that tries to *evolve* a solution over time.

Its first step builds a set of initial solutions, which is called *population*. The aim is to provide diversified solutions, so that the best parts of each individual solution can be preserved over time and favored during evolution.

Another key part is the definition of the *fitness* function (to be maximized), that measures how good a certain individual is. For our problem, the fitness corresponds to the penalized cost of the solution (with sign changed). An important aspect is that, since this fitness function must be evaluated for each new individual at each iteration, computing its cost must be as efficient as possible.

At each iteration of the algorithm, called *generation*, GA creates new solutions in the *crossover* phase. In this step, also called *breeding* phase, the algorithm chooses two solutions from the current population to generate one or more new individuals. The mechanism is to use pairs of existing solutions as parents to generate a child that bears some *chromosomes* from both parents. In our problem this means that the solution tree mixes together parts of the trees of its parents. The new individuals thus generated are added to the population.

The reproduction process in nature is not always perfect, and some mutations can randomly arise and contribute to the evolution of the species. The same concept is applied to the new individuals of the population by allowing for a random mutation of some of their encoding *genes*. This mutation phase has proven to improve the overall performance of genetic algorithms [47].

Since both the crossover and the mutation phase can change an individual solution, sometimes there is the need of a *repair* function to ensure, in our case, that the solution is still a proper tree. This repair phase is very dependent on the *encoding* chosen to represent a solution.

Furthermore, GA exploits a *selection* mechanism, in which the new population is trimmed down. The goal is to preserve the individuals with lower fitness with high probability, while maintaining some of the less fit individuals to avoid reducing the variety of the genetic heritage too much. One important indicator of the quality of the evolution is the average fitness of the population, which should improve over the generations (instead of just monitoring the best "incumbent" individual found).

Finally, the Genetic Algorithm can terminate with a variety of conditions such as a fixed set of generations, a time limit (our choice), a number of generations

since the last improvement or from the last average fitness improvement, etc.

10.1 Tuning

First of all, having a good and varied initial population is important for GA. In our case we decided to include a mix of individuals generated by the GRASP method described in Subsection 4.2. About half of these solutions are then modified by forcing the C turbines closest to the substation to have an arc directly connected to the substation itself. In this way GA is able to better allocate the flows of the arcs connected to the substation—instead of having too few arcs connected to the substation, bearing all the producing power and thus keeping the solution infeasible.

After some preliminary tests, we also chose not to include some `Sweep` solutions in the initial population, because these very good solutions hinder the evolution of the others with worse fitness. Instead, it seems better to gradually add some `Sweep` solutions when the average fitness does not improve for while, since in this way the population has some time to evolve its individuals.

As already mentioned, the fitness function is often the bottleneck of the algorithm. Since the crossover and the mutation phases mix together different solutions, we cannot exploit our parametric cost evaluation and have instead to recompute the costs from scratch. A good improvement in performance has been obtained by pre-computing all crossing arcs: using a look-up table that tells if two arcs cross each other, leads to a factor of about 2 of speedup.

Another crucial aspect for this heuristic is the chosen encoding. For GA a good encoding requires two fundamental properties: *locality*, so that small changes to a solution leads to another one that is close in its solution space, and *heritability*, so the crossover function preserves the substructures of the parent solutions. We compare two different methods to memorize the solution tree: the successor representation and the Prüfer [44] number.

- *Successor*: since the WFCRP trees have only one arc that leaves each turbine, we can represent each solution with an array whose i-th entry contains the tail j of the arc (i, j) leaving turbine i. The only exception is the substation node, which is the successor of itself.

  Using this encoding does not guarantee that our structure is a proper tree: it can have cycles and disconnected components. Therefore there is the need to repair the solutions generated in the crossover and in the mutation phase. This is done by checking for cycles. After removing the arcs that create cycles, we repair the solution by applying the Kruskal's Minimum Spanning Tree algorithm preferring, when possible, those arcs that would not cause a crossing. For this reason we also call this version the Kruskal repair.

- *Prüfer number*: it represents an undirected tree whose nodes are labelled from 1 to n, with a sequence of $n-2$ natural numbers; see [44] for details. Since the Prüfer number always represents a valid tree, with this encoding we do not need a repair step.

On one hand, the Prüfer number has the advantage of being a compact and elegant representation that always encodes a valid tree, while on the other hand it does not have the properties of locality, since changing a single number of its sequence leads to a completely different tree, nor the property of heritability, because when mixing two solutions the substructures of the parents are not preserved. Because of this, the work [32] concludes that using this representation in a GA leads to a poor performance in the general case. The Successor representation satisfies instead the two properties, but has the downside of needing a repair step, which can be computationally expensive.

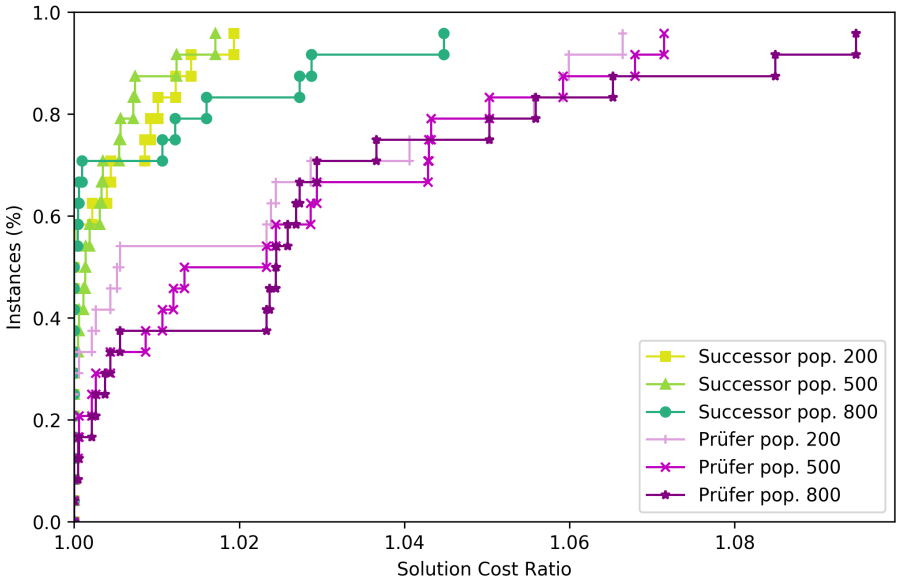

Figure 10: Performance profile for GA on the training instances, comparing the Successor and the Prüfer encoding for population sizes of 200, 500 and 800 individuals each. We can see that the lack of locality and heritability of Prüfer encoding leads to an inferior performance than the Successor encoding for all population sizes tested.

Figure 10 compares the two encoding results (with the same initial solutions and crossover, mutation and selection functions) on the training instances. We observe that it is much better to use the Successor encoding (with Kruskal repair) than the Prüfer numbers, for each population size. For this reason, for our GA we choose the Successor encoding.

We next examine the crossover function. To maintaing a good degree of diversification, we choose two variants, which we pick at random with equal probability: the first one mixes the genetic sequence (that is, the Successor or Prüfer sequence) of the two parents by picking at random each gene (a node number in the tree encoding) either from the father or from the mother. In this scenario we actually build two children solutions: the one as described and its complement, that makes the opposite choice for each gene. The second variant, instead, splits the genetic sequence in half, and generates two children, one with the first half from the father and the second half from the mother, and viceversa for the other child. Each of the newly generated children is added to the current population.

The mutation step is performed with a small probability for all the individuals in the population. If an individual is chosen, each of its genes in the genetic sequence have a small probability of being changed with a random number.

After these two phases, for the Successor representation, the Kruskal repair algorithm is run to ensure they yield valid solutions.

The final step is to select which of the solutions to keep among the ones in the population. To do so we order the solutions by decreasing fitness, scan them and add them to the new population set with high probability, until we reach the maximum population size. In this way we are sure to keep most of the individuals with a good fitness while, at the same time, retaining some of the worse solutions to ensure some genetic variety in the population.

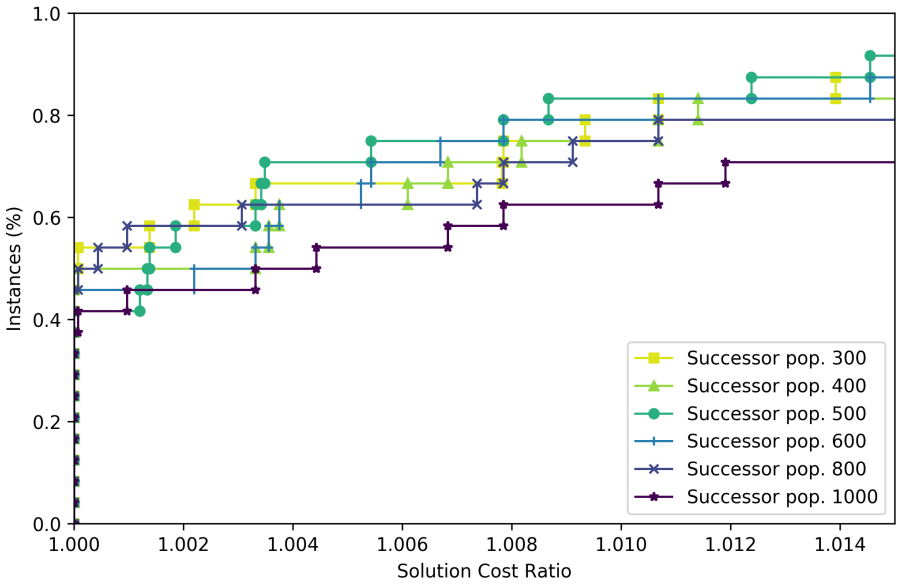

Figure 11: Performance profile comparing population sizes of 300, 400, 500, 600, 800, 1000 of the Genetic Algorithm with Successor encoding; choosing a population size of 500 is slightly better for the hardest instances.

The population size is the last parameter we have to choose, which must be large enough to ensure some diversification of the population but not too much, to avoid including too many bad solutions and make the algorithm slow. In Figure 11 we report the performance profile for the following population sizes: 300, 400, 500, 600, 800, 1000. We can see that 500 individuals is the best choice, probably because it better balances between number of explored generations and solution diversification. Having too many individuals in a population leads instead to a worse performance, as we can see with a population of 1000.

11 Computational experiments

This section analyzes the results of our heuristics on the instances of both the training set and the test set.

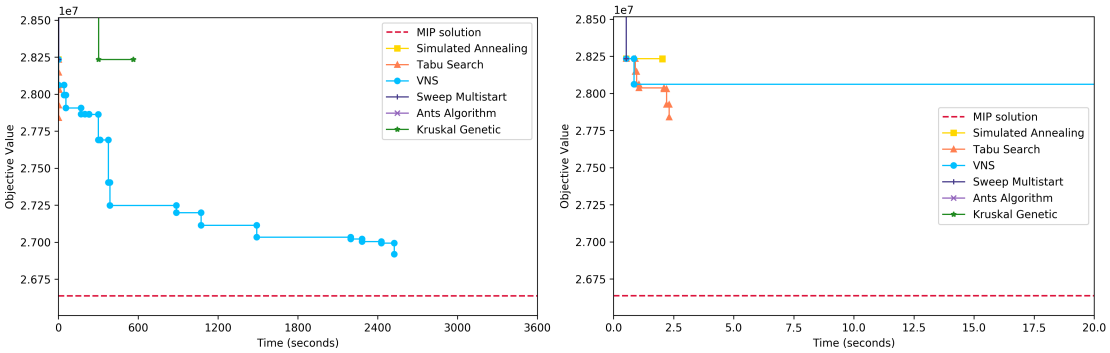

Figure 12: Cost improvement over time, on a training instance with 100 turbine.

All heuristic techniques seldom improve their solutions after a few minutes of computing time, with the only exception of VNS that continues improving, slightly, after the first 10 minutes. Figure 12 plots the evolution of the costs obtained by the heuristics in 1 hour of computing time on training instance n. 28 with 100 turbines. VNS is the only technique which is able to lower the costs after 10 minutes. Simulated Annealing and Tabu Search do not improve their solution after the first few seconds as it does, by design, `Sweep`. The Genetic Algorithm improves the cost until 563 seconds. As to Ant Colony Optimization, it stops improving after 183 seconds but it does not even reach a feasible solution (hence it is not visible in the graph).

For this reason, in what follows a time limit of 10 minutes has been considered, on a Intel® Xeon® CPU E5-2623 v3 at 3.00GHz (quadcore) with 16GB of RAM. The only exception is the `Sweep` heuristic that, in its multistart implementation, is able to try all of its parameter combinations in only a few seconds of computing

time, even for instances with 120 turbines: `Sweep` would not be able to find better solutions in the remaining time, therefore it stops early.

Even though our CPU architecture is multi-core, our heuristics have been implemented as single thread and do not take advantage of the parallelism. This option could be worthwhile to explore, especially for the Genetic Algorithm which could have a gain in performances by exploiting parallel computations when evaluating the cost of the individual solutions in the population.

11.1 Performance on training instances

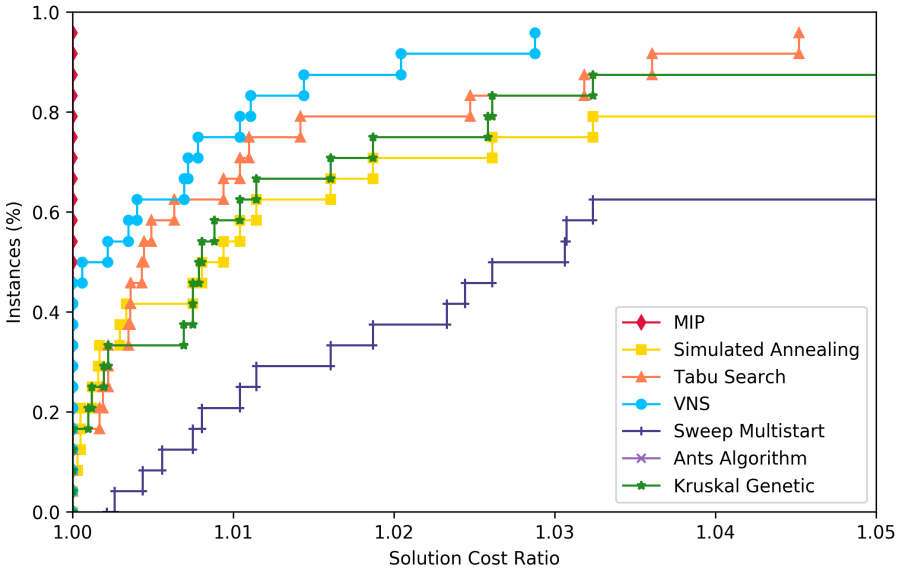

Figure 13: Performance profile on the training instances which compares the MILP-based matheuristic technique results in [26] with the six implemented heuristics. After 10 minutes all heuristics are close to the (near) optimal solutions computed in 1 hour by the matheuristic, except for the Ant Colony Optimization algorithm whose curve lies much further on the horizontal axis and therefore does not even appear in the graph.

Figure 13 compares our heuristics and the MILP-based matheuristic method by [26] on the training set of instances. The matheuristic computes the solutions with a 1-hour limit and uses a commercial MILP solver (IBM ILOG Cplex), reaching provably optimal or near optimal costs in all cases. As expected, the red curve of the MILP-based matheuristic is not beaten by any heuristic technique, since its solutions are mostly optimal. `Sweep` (multistart) solutions are all within 8% of the matheuristic solutions, and requires 1-2 seconds of computing time. Taking into account their short time limit (10 minutes instead of 1 hour), the results of the

heuristics appear quite satisfactory, except for Ant Colony Optimization (which does not start from the `Sweep` solution).

VNS qualifies as the best heuristic technique, reaching the optimal solution in half of the instances (the ones with fewer turbines) and achieving a gap smaller than 3% from the matheuristic solution costs in the other half. Tabu Search is our second-best heuristic, achieving in 5 instances the optimal solution and staying within a 5% gap from the matheuristic for the other training instances. Also the Genetic Algorithm reaches the optimal solution costs in 5 of the training instances, and maintains the gap below 6% from the optimal/best-known costs. Simulated Annealing, instead, is optimal only in 3 cases, and is within the 7.5% from the solution cost of the MILP-based matheuristic; as a matter of fact, it is not able to significantly improve the `Sweep` solutions. Finally, Ant Colony Optimization, as noted before, does not match the `Sweep` solutions in any instance and remains at infeasible solutions in 4 instances, while its feasible solutions have a 30% gap.

11.2 Performance on test instances

We next consider our new test set of 220 instances: Figure 14 shows the results for the heuristic techniques with the substation on top of the layout area and $C = C_{min} + 1$.

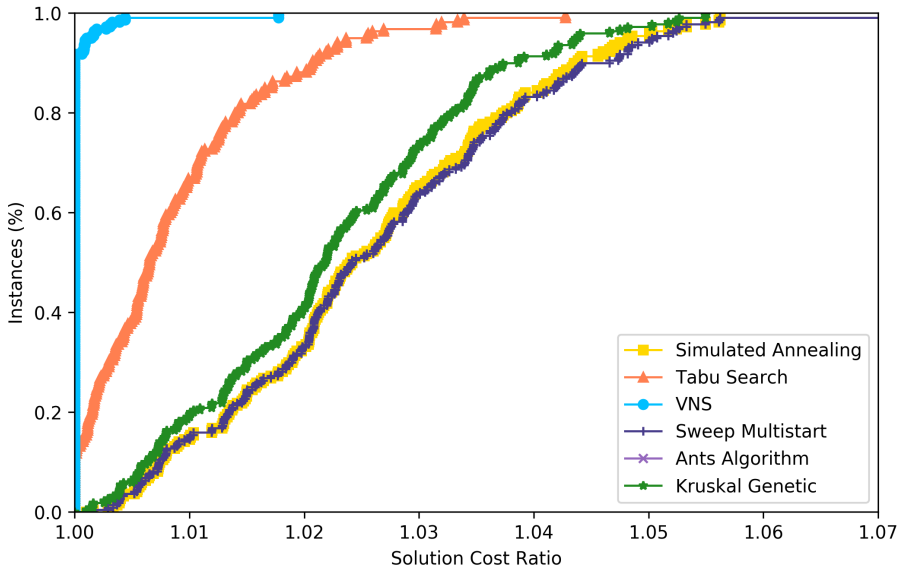

Figure 14: Performance profile on the test instances for the heuristic techniques, with the substation placed in the middle top of the layout and $C = C_{min} + 1$.

The performance profile curves of the heuristics follows the same hierarchy seen on the training instances. In this case the separation between the heuristics is even

more evident.

`Sweep` (multistart) is able to find feasible solutions for all test instances, therefore providing good starting solutions to the other heuristics, whose costs are then improved by about 4-5%.

Variable Neighborhood Search confirms to be the most effective among the heuristics we developed, reaching the best solution among all the six heuristics in 203 out of 220 cases (92.3%) of the. In the remaining 17 cases, it is outperformed by Tabu Search, but only by a small margin: the gap between the VNS solution costs and Tabu Search is less than 0.5% for 16 instances, and 1.77% in just one case.

The second best-performing heuristic is Tabu Search, which has a gap of less than 2% with respect to VNS. It is interesting to note that these two techniques share a local search phase in their design, which is probably the reason why VNS and Tabu Search performs better than the other heuristics: local search allows the heuristic to reach good local minima, while Simulated Annealing and the Genetic Algorithm seem to arrive close to good solutions but have more difficulty to descend to the local minimum point.

The Genetic Algorithm is able to improve the `Sweep` solution only in about 20% of the instances, usually obtaining a cost 1-2% lower. This can be due to two factors: the first is that the internal mechanism of the Genetic Algorithm is more complex and requires heavy computation per generation. Additionally, it can not exploit the parametric cost evaluation technique, so the number of iterations within the time limit is reduced. A second issue is that the Genetic Algorithm, as already mentioned, does not have a proper local search phase.

Simulated Annealing improves the `Sweep` solution in only 8% of the test instances, with an improvement of about 1%. A possible explanation of this behavior is that the test instances have a large number of turbines: Simulated Annealing does not have a systematic local search phase, so for large problems the probability of finding good random moves is low. Furthermore, when the substation is positioned on a wind farm side, the cables connected to the substation are longer and thus it is more likely for a changed arc to create a crossing.

Ant Colony Optimization instead does not even reach feasibility in basically any instance: having the substation on a side of the wind farm leads the algorithm to connect just one or two long cables to it, which are not able to support all the power produced by the wind farm. This issue was not so evident on the training set where the substation is much closer to the turbines, and highlights a critical weakness of the method.

Figure 15 shows the performance profile curves obtained by the heuristics when the `Sweep` heuristic is not applied as a warm start. VNS is the most effective technique even in this case, performing better than the other heuristics which, instead,

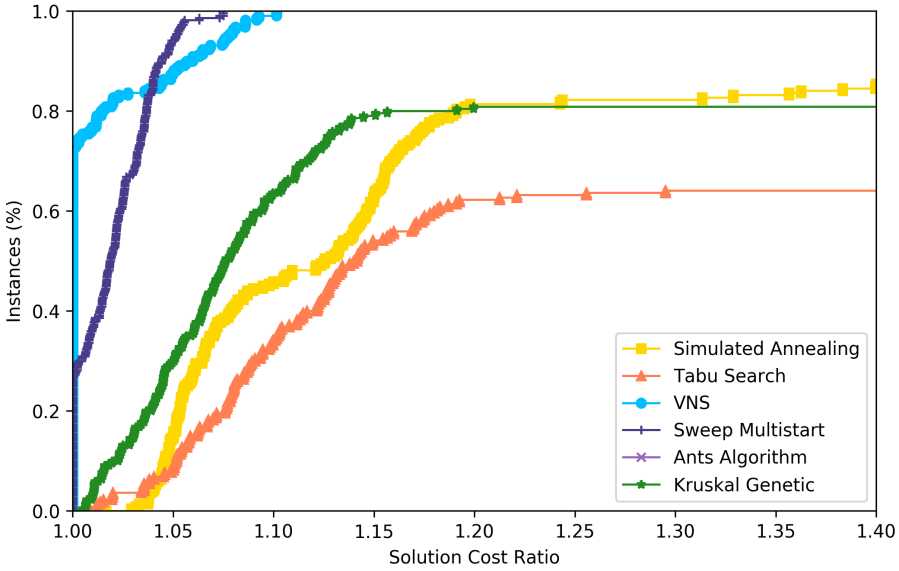

Figure 15: Performance profile on the test instances for the heuristic techniques without starting from the `Sweep` solutions. VNS is the only heuristic that outperforms `Sweep`.

never outperform `Sweep`, confirming the benefits of using the latter to provide good starting solutions. The effectiveness of the other heuristics changes in a more evident way: Tabu Search, in particular, is beaten by both the Genetic Algorithm and Simulated Annealing. This worsening of the Tabu Search performance can be attributed to the many parameters of the algorithm: starting from infeasible solutions may require a different tuning for this heuristic. The Genetic Algorithm performs better than Simulated Annealing in 155 out of 220 cases (70.5%) but in the remaining ones, contrary to Simulated Annealing, it often does not even reach feasible solutions. Ant Colony Optimization, which cannot include `Sweep` solutions by design, remains the worst of our heuristics and is not even visible in the figure.

Figure 16 compares the two versions of VNS: with and without the `Sweep` solution. VNS without `Sweep` is able to reach better costs than VNS with `Sweep` in 20.9% (46 out of 220) of the test instances; however, the improvement is typically less than 1%. Instead, VNS without `Sweep` performs better than `Sweep` itself 72.7% of the times (160 out of 220), but it never reaches better costs on the instances with 120 turbines. These results confirm the effectiveness of `Sweep` and highlight its contribution to the other heuristics: these algorithms, alone, are often unable to reach the solutions that `Sweep` computes in a few seconds.

The performance profile plots remain basically the same when changing the substation position to the left side. This seems to confirm our intuition that the

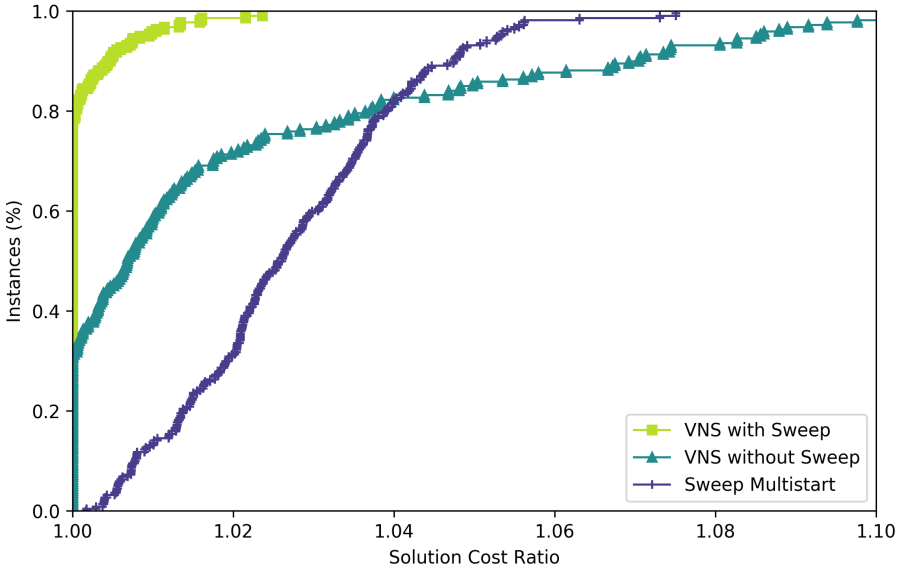

Figure 16: Performance profile on the test instances, comparing two variants of VNS: with and without the starting solution from `Sweep`. VNS performs reasonably well even without a good starting solution, but it is outperformed by `Sweep` alone for the large instances with 120 turbines.

local search phase is crucial for obtaining a good performance for this kind of instances.

When the substation is placed at the geometric center of the layout, instead, the performance profile curves change a bit, as shown in Figure 17. VNS and Tabu Search remain the first and second best heuristics, but the performance of the two algorithms are closer: in 16.8% (37 out of 220) of the instances the two heuristics reach the same solution cost, and 29.5% (65 out of 220) of the times Tabu Search has lower costs than VNS. In the barycentric case, indeed, more turbines are closer to the substation itself, which can also be reached by more arcs without forming any crossing. Furthermore, it is easier for the heuristics to find a good solution, because a central topology favors a good division of the turbines and makes the 1-opt adjustments of these groups easier, having more degrees of freedom in moving turbines between groups to balance the flows. As a matter of fact, the Genetic Algorithm and Simulated Annealing are more often able to improve the `Sweep` solution, and Ant Colony Optimization finds feasible solutions in about half of the test instances.

We also examined how the C parameter, which limits the number of cables that can be connected to the substation, affects the solutions we obtain. We considered cases $C = \{C_{min}, C_{min} + 1, C_{min} + 2, \infty\}$, with the substation positioned at the top of the turbine layouts. The performance profile curves (not reported for the

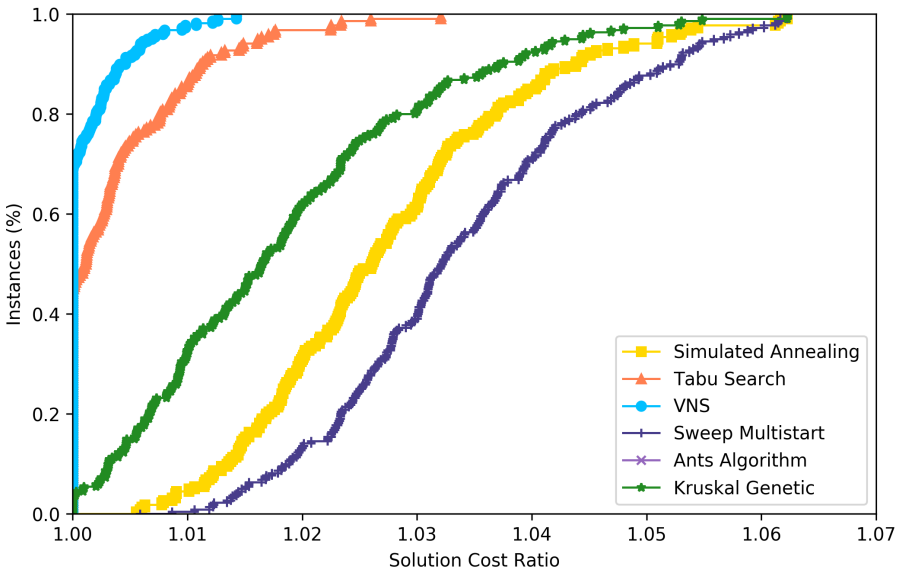

Figure 17: Performance profile on the test set instances with the substation placed in the geometric center of the turbine layout and $C = C_{min} + 1$.

sake of space) are almost identical in the four cases.

The outcome of our computational experience is that some of the developed heuristics have a very good performance, in particular VNS and Tabu Search: both of them reach near optimal solutions for the training instances, and are always the best methods in the test instances. This suggests that exploiting a systematic local-search phase is important to achieve a good performance for WFCRP. VNS is our method of choice, as it very often performs better than Tabu Search and is also easier to implement and tune. Moreover, all the heuristics benefit greatly by starting with (or including) the `Sweep` solutions, which are very fast to compute and turn out to be close to the optimal ones.

As expected, using a central position for the substation makes the optimization easier and can also reduce the overall cable costs. Note however that the substation needs an export cable connected to the shore, so putting it in a central wind farm position likely produces unwanted crossings with the turbine cables and hence obstacles to be avoided; see [20, 21] for a discussion of this topic.

12 Conclusions and future work

The Wind Farm Cable Routing is a relevant problem for energy companies, which can greatly reduce the costs of building a wind farm.

For this problem we have developed six metaheuristic techniques: Sweep, Sim-

ulated Annealing, Tabu Search, Variable Neighborhood Search, Ant Colony Optimization and Genetic Algorithm. Each technique has been tuned on a training set, and then validated on a large new set of 220 test instances that are made publicly available.

Our computational experience has shown that some of the developed heuristics have a quite good performance, in particular VNS and Tabu Search: both of them have reached near optimal solutions for the training instances, and have been always the best methods in the test cases. VNS is our method of choice, as it very often performs better than Tabu Search and is also easier to implement and tune. Moreover, all the heuristics benefit greatly by starting with (or including) our new constructive heuristic, Sweep.

A possible direction for future works is to allow for some crossings in the solutions, accounting for their additional cost by modifying the penalty values in the objective function. Another interesting aspect can be testing the Genetic Algorithm with a multi-thread implementation, and to modify its repair phase to include a sort of local search. Also, an important new feature to implement is the capability of handling obstacles and so-called Steiner nodes.

Finally, it could be worth extending our work to design a heuristic that includes both turbine placement and cable routing optimization, thus allowing for a holistic optimization of a wind farm.

Acknowledgements

We thank Jesper Runge Kristoffersen from Vattenfall AB for providing us with technical expertise and support. We also thank Massimo Dalla Cia who implemented an early version of the Genetic Algorithm. The work of the last author was partially supported by MiUR, Italy (PRIN 2015 project).

References

[1] Bauer, J., Lysgaard, J.: The offshore wind farm array cable layout problem: a planar open vehicle routing problem. Journal of the Operational Research Society **66**(3), 360–368 (2015)

[2] Berzan, C., Veeramachaneni, K., McDermott, J., O'Reilly, U.M.: Algorithms for cable network design on large-scale wind farms. Tech. rep., Tufts University (2011)

[3] Cerveira, A., de Sousa, A., Pires, E.S., Baptista, J.: Optimal cable design of wind farms: The infrastructure and losses cost minimization case. IEEE Transactions on Power Systems **31**(6), 4319–4329 (2016)

[4] Chen, Y., Li, H., Jin, K., Song, Q.: Wind farm layout optimization using genetic algorithm with different hub height wind turbines. Energy Conversion and Management **70**, 56–65 (2013)

[5] Cheriton, D., Tarjan, R.E.: Finding minimum spanning trees. SIAM Journal on Computing **5**(4), 724–742 (1976)

[6] Coffman, E.G., Garey, M.R., Johnson, D.S.: Approximation algorithms for bin-packing—an updated survey. In: Algorithm Design for Computer System Design, pp. 49–106. Springer (1984)

[7] Dorigo, M., Maniezzo, V., Colorni, A.: Ant system: optimization by a colony of cooperating agents. IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) **26**(1), 29–41 (1996)

[8] Dutta, S., Overbye, T.: A clustering based wind farm collector system cable layout design. In: Power and Energy Conference at Illinois (PECI), 2011 IEEE, pp. 1–6. IEEE (2011)

[9] Fagerfjäll, P.: Optimizing wind farm layout: more bang for the buck using mixed integer linear programming. Chalmers University of Technology and Gothenburg University (2010)

[10] Feo, T.A., Resende, M.G.: Greedy randomized adaptive search procedures. Journal of global optimization **6**(2), 109–133 (1995)

[11] Fischetti, M.: Improving profitability of wind farms with operational research. Impact **2019**(1), 30–34 (2019)

[12] Fischetti, M., Fischetti, M.: Matheuristics, *Handbook of Heuristics*, vol. 1-2, pp. 121–153 (2018)

[13] Fischetti, M., Fischetti, M., Monaci, M.: Optimal turbine allocation for offshore and onshore wind farms. In: K. Fujisawa, Y. Shinano, H. Waki (eds.) Optimization in the Real World, pp. 55–78. Springer Japan, Tokyo (2016)

[14] Fischetti, M., Fraccaro, M.: Using OR+AI to predict the optimal production of offshore wind parks: A preliminary study. In: Springer Proceedings in Mathematics and Statistics, vol. 217, pp. 203–211 (2017)

[15] Fischetti, M., Fraccaro, M.: Machine learning meets mathematical optimization to predict the optimal production of offshore wind parks. Computers and Operations Research **106**, 289–297 (2019)

[16] Fischetti, M., Kristoffersen, J.R., Hjort, T., Monaci, M., Pisinger, D.: Vattenfall optimizes offshore wind farm design. Finalist of the 2019 Franz Edelman Award; to appear in INFORMS Journal on Applied Analytics (2019)

[17] Fischetti, M., Leth, J.J., Borchersen, A.B.: A mixed-integer linear programming approach to wind farm layout and inter-array cable routing. In: 2015 American Control Conference, pp. 5907–5912. IEEE (2015)

[18] Fischetti, M., Monaci, M.: Proximity search for 0-1 mixed-integer convex programming. Journal of Heuristics **20**(6), 709–731 (2014)

[19] Fischetti, M., Monaci, M.: Proximity search heuristics for wind farm optimal layout. Journal of Heuristics **22**(4), 459–474 (2016)

[20] Fischetti, M., Pisinger, D.: Inter-array cable routing optimization for big wind parks with obstacles. In: 2016 European Control Conference, ECC 2016, pp. 617–622 (2017)

[21] Fischetti, M., Pisinger, D.: On the impact of using mixed integer programming techniques on real-world offshore wind parks. In: ICORES 2017 - Proceedings of the 6th International Conference on Operations Research and Enterprise Systems, vol. 2017-January, pp. 108–118 (2017)

[22] Fischetti, M., Pisinger, D.: Mathematical optimization and algorithms for offshore wind farm design: An overview. Business & Information Systems Engineering (2018)

[23] Fischetti, M., Pisinger, D.: Mixed integer linear programming for new trends in wind farm cable routing. Electronic Notes in Discrete Mathematics **64**, 115–124 (2018)

[24] Fischetti, M., Pisinger, D.: On the impact of considering power losses in offshore wind farm cable routing, *Communications in Computer and Information Science*, vol. 884 (2018)

[25] Fischetti, M., Pisinger, D.: Optimal wind farm cable routing: Modeling branches and offshore transformer modules. Networks **72**(1), 42–59 (2018)

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

[27] Gendreau, M.: An introduction to tabu search. In: F.W. Glover, G.A. Kochenberger (eds.) Handbook of Metaheuristics, pp. 37–54. Springer (2003)

[28] Glover, F.: Tabu search—part I. ORSA Journal on computing **1**(3), 190–206 (1989)

[29] Glover, F.: Tabu search: A tutorial. Interfaces **20**(4), 74–94 (1990)

[30] Glover, F.: Tabu search—part II. ORSA Journal on computing **2**(1), 4–32 (1990)

[31] Gonzalez-Longatt, F.M., Wall, P., Regulski, P., Terzija, V.: Optimal electric network design for a large offshore wind farm based on a modified genetic algorithm approach. IEEE Systems Journal **6**(1), 164–172 (2012)

[32] Gottlieb, J., Julstrom, B.A., Raidl, G.R., Rothlauf, F.: Prüfer numbers: A poor representation of spanning trees for evolutionary search. In: Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation, pp. 343–350. Morgan Kaufmann Publishers Inc. (2001)

[33] Hansen, P., Mladenović, N.: Variable neighborhood search. In: F.W. Glover, G.A. Kochenberger (eds.) Handbook of Metaheuristics, pp. 145–184. Springer (2003)

[34] Hertz, A., Marcotte, O., Mdimagh, A., Carreau, M., Welt, F.: Design of a wind farm collection network when several cable types are available. Journal of the Operational Research Society **68**(1), 62–73 (2017)

[35] Hou, P., Hu, W., Chen, Z.: Optimisation for offshore wind farm cable connection layout using adaptive particle swarm optimisation minimum spanning tree method. IET Renewable Power Generation **10**(5), 694–702 (2016)

[36] Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science **220**(4598), 671–680 (1983)

[37] Klein, A., Haugland, D.: Obstacle-aware optimization of offshore wind farm cable layouts. Annals of Operations Research pp. 1–16 (2017)

[38] Kusiak, A., Song, Z.: Design of wind farm layout for maximum wind energy capture. Renewable Energy **35**(3), 685–694 (2010)

[39] Ledesma, S., Aviña, G., Sanchez, R.: Practical considerations for simulated annealing implementation. In: C.M. Tan (ed.) Simulated Annealing, chap. 20. IntechOpen, Rijeka (2008)

[40] Li, D.D., He, C., Fu, Y.: Optimization of internal electric connection system of large offshore wind farm with hybrid genetic and immune algorithm. In: Proceedings of the Third International Conference on Electric Utility Deregulation and Restructuring and Power Technologies, pp. 2476–2481. DRPT 2008 (2008)

[41] Maples, B., Saur, G., Hand, M., van de Pieterman, R., Obdam, T.: Installation, operation, and maintenance strategies to reduce the cost of offshore wind energy. Tech. rep., National Renewable Energy Lab. (NREL), Golden, CO (United States) (2013)

[42] Mladenović, N., Hansen, P.: Variable neighborhood search. Computers & Operations Research **24**(11), 1097–1100 (1997)

[43] Nedlin, J.: Ant-based algorithms for the wind farm cable layout problem. Ph.D. thesis, Karlsruhe Institute of Technology (2017)

[44] Palmer, C.C., Kershenbaum, A.: Representing trees in genetic algorithms. IBM Thomas J. Watson Research Division (1994)

[45] Pillai, A., Chick, J., Johanning, L., Khorasanchi, M., de Laleu, V.: Offshore wind farm electrical cable layout optimization. Engineering Optimization **47**(12), 1689–1708 (2015)

[46] Samorani, M.: The wind farm layout optimization problem. In: P.M. Pardalos, S. Rebennack, M.V. Pereira, N.A. Iliadis, V. Pappu (eds.) Handbook of Wind Power Systems, pp. 21–38. Springer (2013)

[47] Srinivas, M., Patnaik, L.M.: Genetic algorithms: A survey. IEEE Computing **27**(6), 17–26 (1994)

[48] Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing **1**(2), 146–160 (1972)

[49] Whitley, D.: A genetic algorithm tutorial. Statistics and Computing **4**(2), 65–85 (1994)

[50] WWEA: Wind power capacity reaches 539 gw, 52,6 gw added in 2017 (2018). URL `http://wwindea.org/blog/2018/02/12/2017-statistics/`. Online; accessed 20-November-2018

[51] Zhao, M., Chen, Z., Blaabjerg, F.: Optimisation of electrical system for offshore wind farms via genetic algorithm. IET Renewable Power Generation **3**(2), 205–216 (2009)

[52] Zuo, T., Meng, K., Tong, Z., Tang, Y., Dong, Z.H.: Offshore wind farm collector system layout optimization based on self-tracking minimum spanning tree. International Transactions on Electrical Energy Systems p. 2729 (2018)