Karlsruhe Institute of Technology

Simulated Annealing-Based Heuristics for Wind Farm Cabling Problems

Master's Thesis of

Sebastian Lehmann

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. rer. nat. Peter Sanders
Advisors:	Dr. rer. nat. Ignaz Rutter
	Franziska Wegner

Time Period: 1st December 2015 – 11th May 2016

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, May 20, 2016

Abstract

In a wind farm, the cables connecting turbines and substations with the power grid significantly contribute to its installation cost. Minimizing the cabling cost traces back to a capacitated network flow problem similar to those arising in areas such as logistics and transport. In this work, we study the following cable layout problem: In the wind farm, the locations of turbines and substations are fixed, and they should be connected with cables. We are given a set of cable types, which differ in their cost per meter and in the amount of power they support. We are interested in the cable network minimizing the cabling cost, such that the farm is connected properly.

We formally model this problem as a capacitated minimum-cost flow problem with non-convex edge cost functions. This problem is NP-hard which means that, as matters stand, no polynomial time algorithm is known. We also model the cable layout problem as a mixed integer linear program (MILP) which can be solved using a generic optimizer. As an alternative, we propose a heuristic based on Simulated Annealing, which we qualitatively compare with the MILP solved with Gurobi Optimizer, a generic optimization program.

In our experiments, it turns out that our heuristic is a good alternative to the MILP model for small and medium-sized instances. For wind farms with up to 450 turbines, our solution outperforms the results of Gurobi in the majority of cases. We propose different modifications to our heuristic, of which only some were useful to achieve these results. We conclude the work with different additional ideas for improvements which could be tried to make the heuristic suitable for even larger wind farms.

Deutsche Zusammenfassung

Die Konstruktionskosten eines Windparks hängen unter anderem davon ab, wie die Windräder und die Umspannwerke miteinander verkabelt sind. Es ist daher von großem Interesse, die Kosten für diese Verkabelung auf ein Minimum zu beschränken, ohne die gegebenen Vorausetzungen zu verletzen. Dabei stehen uns mehrere Kabeltypen zur Verfügung, welche für unterschiedliche Ströme ausgelegt sind und verschiedene Konstruktionskosten verursachen.

Das Verkabelungsproblem lässt sich auf ein kapazitives Flussproblem reduzieren, welches auch für Transportprobleme wie zum Beispiel in der Logistik Anwendung findet. Auch dort hat man verschiedene Transportmittel zur Auswahl, welche sich in ihrer Kapazität und den Kosten unterscheiden. In beiden Szenarien verhalten sich die Kantenkosten dabei nicht-konvex, was das Problem NP-schwer macht. Nach dem aktuellen Stand der Forschung ist demnach kein Algorithmus bekannt, welcher dieses in polynomieller Laufzeit löst. Gleichzeitig lässt sich das Verkabelungsproblem auch als gemischtes ganzzahliges lineares Programm (engl.: mixed integer linear program, MILP) formulieren. Dieses ist zwar immer noch NP-schwer, jedoch lässt es sich mit einer generischen Optimierungssoftware wie Gurobi approximativ lösen.

Als Alternative schlagen wir eine Heuristik vor, welche auf Simulated Annealing, einer physikalisch motivierten Optimierungsheuristik, basiert. Wie wir in Experimenten zeigen, werden damit Ergebnisse erreicht, welche gegen diejenigen des Gurobi Optimizer für kleine und mittelgroße Windparks konkurrieren kann. Für bis zu 450 Turbinen erreicht unsere Heuristik bessere Ergebnisse in kürzerer Zeit. Wir stellen Verbesserungsvorschläge vor, mit welchen wir diese Ergebnisse erzielt haben. Wir schließen die Arbeit mit weiteren Vorschlägen ab, welche auch für noch größere Windparks Anwendung finden könnten.

Contents

1 Introduction **1**

2 Related Work **5**

3 Preliminaries **9**
3.1 Flow Networks 9
3.2 Simulated Annealing 11

4 Optimization Problems in Wind Farms **15**
4.1 The Cable Layout Problem 15
4.2 The Substation Assignment Problem 20
4.3 The Substation Positioning Problem 21

5 Heuristics for Wind Farm Problems **23**
5.1 An Algorithm based on Simulated Annealing 23
5.2 Improving the Algorithm 26
5.3 Towards Evolutionary Algorithm 29
5.4 Improvement using a Two-Level Approach 31

6 Experiments **35**
6.1 Generating Random Instances 38
6.2 General Observations 41
6.3 Analyzing the Behavior of the Basic Algorithm 48
6.4 Evaluation of Our Suggested Improvements 51

7 Conclusion **59**

Bibliography **61**

Appendix **63**

1. Introduction

A wind farm is a kind of collector system which converts kinetic energy in the form of wind to electrical energy, which is injected into the power grid (see Figure 1.1). For this, wind turbines play the major role in this energy conversion, however they cannot be directly attached to the power grid due to several reasons.

First, to produce a considerable amount of power, a typical wind farm consists of many turbines. They are either placed on a field, in which case we call it an *onshore* wind farm, or on the sea, which is called an *offshore* wind farm. Connecting all of them separately to the power grid would require many cables over long distances. Instead, the power is routed through a network of cables connecting the turbines to a grid point, which builds the access to the (high voltage) power grid.

Turbines deliver their power as alternating current (AC) in a relatively low voltage of around 33 kV, while the electricity grid usually operates at 110 kV, 220 kV or 380 kV. Transporting electrical energy over cables is always subject to power loss, which is greater for lower voltage levels and longer distances. Therefore, when connecting many turbines together and when transporting the generated power over longer distances like in an offshore farm, high voltage direct current (HVDC) is recommended to reduce the power loss. Also, for really long distances in the range of hundreds of kilometers, AC is subject to inductive and capacitive power loss. This is the case for offshore farms which are very distant from the shore. In these cases, it might be beneficial to also convert the power into high-voltage direct current using an HVDC converter. Whenever transformers are required, they are built into a so-called *substation* near the turbines. From there, *export cables* deliver the power to a grid point. There, yet another transformer might be installed to convert the power back into alternating current and to the voltage level of the electrical grid in which it is finally injected.

For example, the BARD Offsore 1 Wind Farm, as shown in Figure 1.2, is a German wind farm in the North Sea, which was built in 2010. It consists of 80 turbines palced on a region of 59 square kilometers, each with a maximum power rating of 5 MW. They add up to a total maximum rating of 400 MW for the whole project. Each turbine has its own transformer, exporting the power in the form of 33 kV AC. They are connected to a substation in 10 circuits of 8 turbines each. The substation has two transformers, converting the 33 kV to 150 kV for export to BorWin 1 converter, another substation near the wind farm [4coa, bar, 4cob].

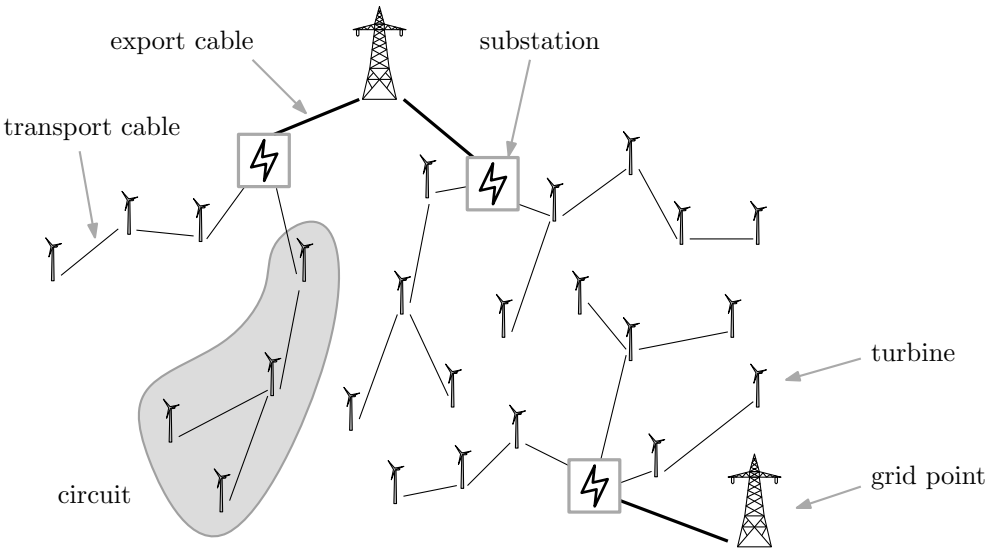


Figure 1.1: Components in a typical wind farm: turbines produce power, which is delivered to a substation by a network of transport cables connecting multiple turbines with each other. Export cables connect the substations to the power grid.

When designing such a wind farm, a lot of decisions have to be made, influencing the building and operating costs. This includes the type, the number and the location of the turbines and the substations, as well as where and which types of cables are used to connect them to the grid points.

Designing a small wind farm with minimal building costs and maximum operating efficiency can be done manually, as the number of possible combinations of choices is quite manageable. However, today's wind farms are already very large and new projects tend to get even larger, so that planning them by hand is no longer reasonable. The demand for tools automating the design decisions is growing. For example, there is *Openwind*, "a wind project design and optimization software that provides professional wind developers with the tools they need to design, analyze and optimize a wind farm" [ope].

The design problems have in common that simply brute-forcing all combinations of choices takes too much time even with today's computing power, as their number rises exponentially with the number of turbines. Such an approach would likely incur costs for computation and project delay which would exceed the savings the solution could ever provide.

In this work, we focus on the cabling problem. We consider a given number of turbines, each with a given location, as well as a given set of cable types we can use in the network for the transportation cables. First, also the substations have fixed locations and support a given number of turbines, while not considering the export cables at all. Then, we relax this problem by not fixing the locations of the substations. We are then interested in their optimal position which minimizes the total cost for the transport and export cables of the wind farm.

The work is structured as follows. In the following chapter, we outline the related work, and in Chapter 3 we introduce some basic preliminaries. The problem is modeled formally in Chapter 4 as both a flow network problem and as a formulation of a mixed integer linear program (MILP). Our main contribution however is an alternative algorithm which we introduce in Chapter 5, a heuristic based on Simulated Annealing. In Chapter 6, we evaluate our idea and compare its quality performance to the MILP as solved by the generic solver *Gurobi Optimizer* [gur]. A conclusion in Chapter 7 completes this work.

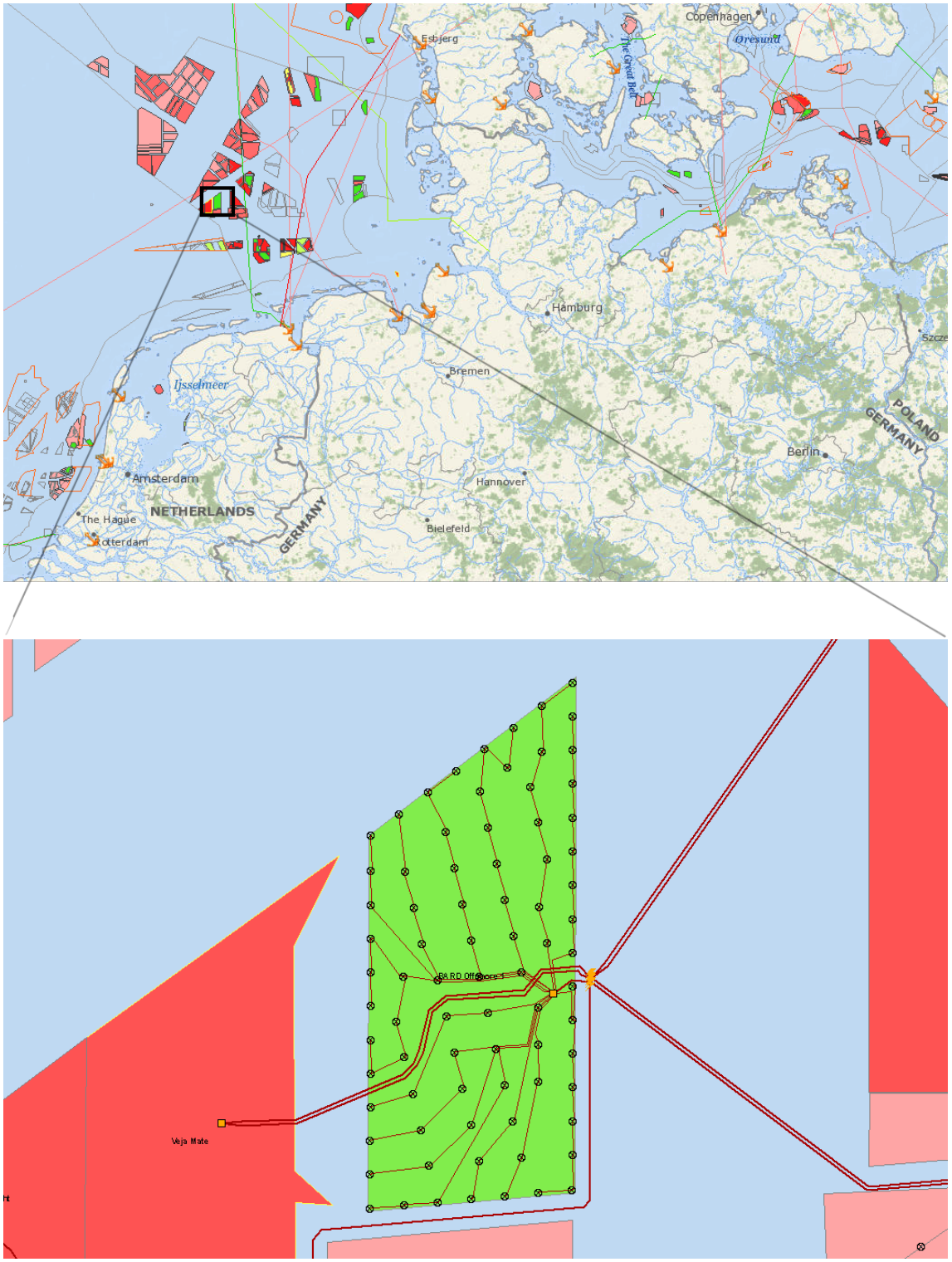


Figure 1.2: Screenshots from the *Global Offshore Wind Farm Map* by 4C Offshore [4coc], showing the BARD Offshore 1 Wind Farm. The nearby red shaded wind farms are projects under construction, and the pink ones are being planned (as of May 2016). The first map shows the north of German and the North Sea. The second image is a zoomed-in view of BARD Offshore 1. We see its turbines, its own substation, the near BorWin 1 converter station as well as cables interconnecting the sites.

2. Related Work

In this chapter, we give a brief overview of related work. Problems similar to ours are found in the literature, but they exist in different variants. The work most related to ours, since we adopted their problem model, is [Ber]. They propose several solutions for wind farm cable layout problems on different scales, which we summarize in the first section. In the second section, we give a brief overview of a variant of our cable layout problem as proposed in [HCB06]. They solve their problem with an Integer Linear Program with different kinds of optimizations. Finally, we present a work which solves a similar problem found in the field of transportation using the meta-heuristic Simulated Annealing [YA12], on which our heuristic is based on.

Wind Farm Cabling Problems

In [Ber], Berzan et al. formulate a cabling problem on three levels. First, they consider a Circuit Problem, in which only the inter-turbine cabling is considered, up to a single turbine which is finally connected to a substation. Their Substation Problem considers multiple of these circuits, all connecting to the same substation. Finally, in the Full Farm Problem multiple substations are taken into account.

In all of their problem variants, they assume fixed positions for all given turbines and substations. They assume the cabling cost to depend on the terrain, which is modeled by a 2D grid containing cost factors, which is laid over the farm's bounding rectangle.

They suggest the following algorithms when they restrict the problem to a single cable type (in general, these problems then nicely map to graph problems): The Circuit Problem is solved by finding the Minimum Spanning Tree (MST). No constraints have to be considered at all, since the cable connecting the circuit to the substation already has to support all the power generated by the turbines of the circuit. Since there is only one cable type, every internal cable of the circuit also satisfies the limits.

Similarly, they solve the Substation Problem with a Capacitated Minimum Spanning Tree (CMST), that is MST with an additional capacity constraint: no sub-tree of the root may contain more than k nodes. They set the capacity k to the cable's limit to solve the Substation Problem. However, CMST is NP-hard [JR05], making it impracticable for large wind farms. However, polynomial time heuristics exist, for example the Esau-Williams heuristic [JR04, JR05].

For the Full Farm Problem, they describe a two-level approach. First, they assign turbines to substations by solving a special kind of min-cost flow network. Then, they approximately

solve the multiple substation problems with the Esau-Williams heuristic. However, in the first step, they approximate the cost for assigning a turbine to a substation by the cost for laying a cable directly. But the actual cost depends on the rest of the network.

They introduce solutions when they relax the problem to multiple cable types. In general, the problems do not map nicely to well-known graph problems anymore. For the Circuit Problem, they introduce three solutions: an MILP, a Backtracking algorithm and a Divide-and-Conquer algorithm, which we briefly describe in the following. However, they do not provide solutions for the Substation and Full Farm Problems.

As their first solution, they model the problem using mixed integer linear programing (MILP): They use boolean variables to decide which cable types to use for each edge (they use one variable for each node pair and cable type). Additionally, they use an integer helper variable for each edge to track the flow on that edge. The size of the MILP grows quadratically in the number of turbines. In general, MILPs are NP-hard [GJ79], and their results perform really poorly: instances with more than 8 turbines have impractical running times. They do not say how much time the computation takes exactly. They use the open-source MILP optimizer library *lp-solve* [lp-].

A second algorithm is based on backtracking: a spanning tree is built recursively by adding a cable to the current tree, until a tree for the whole circuit is constructed (which is then a solution candidate), or until the current tree is already worse than the best solution candidate seen so far (in which case they reject the branch). They introduce a heuristic, giving a lower bound for a sub-problem, making it possible to reject poor branches very early. Although their approach seems quite promising, it is not really an improvement over the MILP, as it only solves problem instances with up to 10 turbines. Again, they do not say how much time the computation takes.

Finally, they introduce a Divide-and-Conquer algorithm. The set of nodes is partitioned recursively in every way possible, and each resulting sub-problem is being solved. Due to the high complexity of such a recursion tree, they also use the lower-bound-heuristic from the backtracking algorithm to reject poor exploration branches early. Furthermore, they use a cache to avoid multiple evaluations of the same sub-problem in different exploration branches. From their tree solutions, this is the one which performs best, but it is still quite slow: finding the optimal cabling for 14 turbines takes less than a minute, but the algorithm scales exponentially. They only solved instances with up to 18 turbines, for which their algorithm takes about 500 seconds in average to complete.

Optimized Linear Programming

In [HCB06], Hertz et al. introduce a similar problem in which the power is unsplittable: at each turbine, the incoming power on a cable must leave the turbine through another single cable, and can not be split among multiple outgoing cables. However, they allow a turbine to have multiple outgoing cables, but the power of each incoming cable is mapped to a distinct outgoing cable. They model the problem as an MILP in which they use binary variables for every pair of cables, of which one is coming in and the other is going out of the same turbine. For every incoming cable, only one such variable may have the value 1. They solve their MILP with CPLEX [cpl]. For instances with 30 to 80 turbines, they achieve objective gaps in the one-digit percentage range after one hour.

They introduce an improvement of the MILP by adding cutting planes as tightening constraints to their model. Several different kinds of constraints are suggested, which they evaluate separately. When using the constraints which result in the best acceleration, they were able solve a real life instance from an industrial partner consisting of 112 turbines with only 6 percent gap after one hour computing time, and 2 percent after 48 hours.

Simulated Annealing

Although the approaches based on Linear Programming seem very promising, our goal was to find an alternative algorithm to heuristically solve wind farm problems. They nicely map to flow problems, which are thoroughly explored in the field of graph theory. However, it is known that if the edge costs are not of a convex shape (which is the case here, as we will see in Chapter 5), the flow problem is NP-hard in general. Similar to MILPs, approximations and heuristics for this kind of problems are used to solve them non-exactly. Different kinds of heuristics for non-convex flow problems have been researched, but most of them are specialized for specific problems.

Our problems have a lot in common with transportation of goods: In our case, the cost for laying a cable does not depend on the actual amount of power it is transporting. However, when exceeding the maximum power the cable is designed for, a different cable has to be used, which usually is more expensive. Similarly, when planning transportation of goods by trucks from many producers to many consumers, the cost for deploying a truck only slightly depends on its load. When the quantity of goods to be transported on a route exceeds the capacity of the truck, a second one has to be deployed, raising the cost in a non-convex manner.

In the field of logistics, one problem which arises is the Multicommodity Capacitated Network Design (MCND) Problem [YA12]. It has many applications, such as designing an optimal transportation plan for delivering goods from production sites to consumer sites. Here, multiple commodities (goods) are to be delivered from multiple producers to multiple consumers (each with a given amount of production or consumption). The edges in the network have capacity limits and two kinds of costs: a base cost for using an edge, and a cost per unit. This results in an edge cost function of a non-convex shape because of the change from zero to the first unit to be transported. The non-convex cost functions make this problem NP-hard [YA12].

Different kinds of heuristics have been developed to solve this kind of problem. To name a few, there is Tabu Search [GL99], Ant Colony Optimization [Dor01], and Simulated Annealing [OL96]. We decided to focus on the latter and adapt it to our problem definition, as results from many research works look quite promising. The Simulated Annealing heuristic is a randomized Monte Carlo algorithm which we explain in detail in Chapter 3. Basically, an initially random solution is modified iteratively. After each modification, if the new objective value is better than before, it is accepted as the new solution for the next iteration. However, if it is worse than before, it is only accepted with some probability, while the probability of accepting depends on the objective difference and on how long the algorithm is running: in the beginning, solutions which are a lot worse are accepted, while towards the end, solutions which are only slightly worse are accepted unlikely.

For example, Yaghini et al. [YA12] solve the MCND Problem using Simulated Annealing on which our approach is based. For each commodity, they only consider a single producer and a single consumer node in a graph. For k commodities and n nodes, they define a $k \times n$ matrix, in which each entry (i, j) represents the priority of node j for commodity i. This indirectly represents a flow: the matrix is decoded into a path for each commodity. Starting from the producer, the next node in the path is determined by the neighbor with maximum priority for that commodity, while cycles are avoided by restricting the neighborhood to the nodes not yet in the path under construction. The objective function (transportation cost) is then computed as a sum of edge costs, of which each is the sum of flows multiplied by the cost-per-unit, plus the edge's opening cost if its flow is not zero.

They compare their algorithm against a Linear Programming (LP) model solved with CPLEX. They use instances with 20 to 30 nodes. When the number of commodities is

low (e.g. 40), CPLEX finds the optimal solution quite fast (under one minute), while their algorithm takes much longer. However, for more commodities (e.g. 200), CPLEX did not find an optimal solution within 10 hours before they stopped the computation. Their algorithm delivers near-optimal solutions in about 8 minutes with an objective gap of 15 to 25 percent. However, they do not present the objective gap for longer running times of their algorithms, or the CPLEX gap for a running time of 8 minutes, so that a fair comparison cannot be made. Yet their approach looked promising to us and serves as the basis of our algorithm.

3. Preliminaries

We define the following notations for sets of numbers. As commonly, the set of *real numbers* is denoted by $\mathbb{R}$. With $\mathbb{R}_{\geq 0}$, we mean the set of *non-negative real numbers* including zero. Intervals of real numbers including the boundaries are denoted with square brackets: $[a, b] := \{x \in \mathbb{R} : a \leq x \leq b\}$. When using parenthesis, the boundaries are excluded: $(a, b) := \{x \in \mathbb{R} : a < x < b\}$. Mixed forms are also possible: $[a, b) := \{x \in \mathbb{R} : a \leq x < b\}$.

The set of *integers* is denoted by $\mathbb{Z}$, and likewise we denote the set of *non-negative integers* as $\mathbb{Z}_{\geq 0}$. The set of *natural numbers* is denoted by $\mathbb{N}$, which does not include zero. To denote the first k natural numbers, usually used for indexing a family of k variables, we use the symbol $\mathbb{N}_{\leq k} := \{1, \ldots, k\}$.

3.1 Flow Networks

In general, a (simple) *graph* $G = (V, E)$ is an abstract structure representing a set of objects, which we call *nodes*, and pairwise connections between them, which we call *edges*. The set of nodes is denoted as V and the set of edges as E. When the graph is *directed*, the edges have a particular direction. For the two nodes $v, w \in V$, the edge connecting them is denoted as the pair $(v, w) \in E$. It is distinct from the opposite edge $(w, v) \neq (v, w)$, although both can be in E at the same time. When the graph is *undirected*, edges do not have a particular direction. In this case, an edge connecting v and w is written as the set $\{v, w\} \in E$.

We introduce the *neighborhood* $N(v)$ of $v \in V$ as the set of *adjacent* nodes that are connected to v via an edge. These edges are called *incident* to v and are denoted as $\delta(v)$. Note that in the case of a directed graph, these sets include both incoming and outgoing edges incident to v. To make a distinction of these two cases, we split $\delta(v)$ in the sets of incoming edges $\delta_-(v)$ and outgoing edges $\delta_+(v)$. Similarly, the neighborhood $N(v)$ is split into $N_-(v)$ and $N_+(v)$, which denote nodes connected to v via incoming or outgoing edges, respectively.

In a directed graph $G = (V, E)$, a *flow* $f : E \to \mathbb{R}$ assigns each edge a *flow value*. When it is negative for an edge $e = (v, w)$, it represents a flow in the edge's opposite direction, that is from w to v, instead of from v to w. When dealing with undirected graphs as the original input in an algorithm using a flow, it is common to define a corresponding directed graph, in which the edge directions are chosen arbitrarily but fixed throughout the algorithm, to have a notation of flow direction.

Given the flow f, the *excess* $\mathrm{ex}(v)$ of node $v \in V$ is defined as the sum of incoming minus outgoing flow over all incident edges:

$$\mathrm{ex}(v) := \underbrace{\sum_{e \in \delta_-(v)} f(e)}_{\text{incoming into } v} - \underbrace{\sum_{e \in \delta_+(v)} f(e)}_{\text{outgoing from } v} \tag{3.1}$$

Observe that the sum of excesses of all nodes is always zero for any f, that is $\sum_{v \in V} \mathrm{ex}(v) = 0$. This is because the flow value of every edge in the graph contributes to the whole sum once positively and once negatively.

In the following, we define a flow network problem, in which a flow is to be found under certain constraints. Note that different kinds of definitions for flow networks exist in the literature, and in the following we describe the one we use in our algorithms[1]. The graph $G = (V, E)$, the function $u : E \to \mathbb{R}_{\geq 0}$ which assigns each edge $e \in E$ a *capacity* $u(e)$, and the function $b : V \to \mathbb{R}$ which assigns each node $v \in V$ a *balance* $b(v)$, form the *flow network* (G, u, b). Depending on $b(v)$, a node is a *source* (also called producer) when $b(v) < 0$, or a *sink* (also called consumer) when $b(v) > 0$. For nodes which are neither sources nor sinks, $b(v) = 0$. The corresponding *flow problem* asks for a flow from sources to sinks under two kinds of constraints, resulting from u and b.

A flow f is a *feasible* solution to the flow network (G, u, b), if for all edges $e \in E$ the absolute value of the flow $|f(e)|$ does not exceed the edge's capacity $u(e)$. We call this the *edge capacity constraint*:

$$-u(e) \leq f(e) \leq u(e) \tag{3.2}$$

Additionally, f has to satisfy the *flow conservation constraint* for all nodes $v \in V$, which states that its excess shall be equal to the given balance:

$$\mathrm{ex}(v) = b(v) \tag{3.3}$$

Minimum-Cost Flow Problem

The above flow problem (G, u, b) is a decision problem: there either exists a feasible flow $f : E \to \mathbb{R}$, or the network is infeasible. We formulate a corresponding optimization problem, the *minimum-cost flow problem* $(G, u, b, c_{(\cdot)})$, asking about an optimal solution, if any exists.

For this, the additional parameter $c_{(\cdot)}$ in a problem instance specifies for any edge $e \in E$ a corresponding edge cost function $c_e : \mathbb{R} \to \mathbb{R}_{\geq 0}$. Given a flow f, the non-negative cost of edge e is given by $c_e(f(e))$. For symmetry reasons, we require that the cost for a negative flow equals the cost for the corresponding positive flow, that is $c_e(-x) = c_e(x)$ for all $x \in \mathbb{R}$.

The objective function which is to be minimized is the total cost $c_{\text{total}}(f)$, that is the sum of all edge costs for a flow f:

$$c_{\text{total}}(f) := \sum_{e \in E} c_e(f(e)) \tag{3.4}$$

[1] In this work, we use a many-sources many-sinks network, but the most common definition of flow networks used in the literature has a single source and a single sink node. However, the former can easily be transformed into the latter by adding a dummy source and a dummy sink, which are connected to all sources and sinks, respectively.

The structure of the edge cost functions is essential for the complexity of the flow network problem. If they are linear functions, then the optimization problem is solvable in polynomial time [Min86, GT88]. The same holds for instances in which the cost functions are contiguous and convex functions composed of several linear segments. Such an instance can be transformed into an instance with just linear cost functions by copying the edges such that every copy of the input edge models a linear segment of the original cost function.

As we will see later, the cost functions we deal with are non-convex. In this case, the flow network problem is NP-hard. This is because we can solve an instance of the Capacitated Minimum Spanning Tree (CMST) problem, which is known to be NP-hard [JR05], with an algorithm solving the flow network problem as explained above. We migrate the input graph as well as the edge capacities of the CMST instance into our model. We set the balance of the root node of the CMST instance to $n - 1$ (where n is the number of nodes in the CMST instance), while setting it to -1 for all other nodes. Furthermore, we set the cost function of all edges to 0 for a flow value of 0, and for any non-zero flow we set it to the edge cost in the CMST instance. The total cost of our flow network corresponds to the cost of the CMST, and the edges which have a non-zero flow value are the spanning tree we search for.

3.2 Simulated Annealing

Optimization problems can be solved exactly or, whenever an exact solution takes too much time to be computed, approximated or solved heuristically. In general, the solution of a heuristic algorithm does not perform as well as the exact solution, when compared by their cost. However, they usually outperform exact solvers in their computation time a lot, especially when the problem is NP-hard.

In general, a heuristic solver usually searches for a good solution in the solution space in a stochastic way. Simulated Annealing is one such heuristic, and is the base of our algorithms in this work. Its idea comes from the physical process of cooling down a glowing metal: its molecular structure can be altered while being hot enough. When cooling it down very quickly, its structure is frozen, not giving it any time to get into an energy-optimal form. However, when cooling the metal slowly, molecules still have time to move around such that the energy of the material is minimized: a crystal structure is formed. In nature, molecules move around randomly, and the material can have an intermediate state with higher energy before getting into a state with better energy. The probability for this to happen decreases with decreasing temperature.

This is the core idea of Simulated Annealing. The overall algorithm for an abstract problem is shown in Algorithm 3.1. Given an instance I, we first create an initial solution R by the initialization function (line 2). It is repeatedly changed slightly over time by the mutation function (line 5). The energy of a solution is determined by an evaluation function (line 3, line 6). Whenever a change results in lower energy, it is accepted as the new solution candidate (line 8). However, when it raises the energy, it is only accepted with some probability. The probability depends on a temperature parameter: the lower the temperature, the less the probability of accepting a worse solution (line 10). Also, the probability depends on how much worse the new solution is. The temperature is lowered over time (line 14).

Before using Simulated Annealing to solve a concrete optimization problem, we have to define a representation of a solution, the corresponding evaluation function, as well as an initialization and a mutation function.

The *representation* is an indirect way to fully describe a solution candidate. Its structure may be very different from the solution itself, but deriving it from a representation needs

Algorithm 3.1: Simulated Annealing

Input: Instance I, Initial temperature T_0, Cooling speed τ
Output: Near-optimal solution R of I

$T \leftarrow T_0$
$R \leftarrow \textsc{Init}(I)$ `/* initial solution */`
$E \leftarrow \textsc{Eval}(I, R)$
repeat
  $R' \leftarrow \textsc{Mutate}(I, R)$ `/* new solution candidate */`
  $E' \leftarrow \textsc{Eval}(I, R')$
  if $E' \leq E$ **then**
    $a \leftarrow 1$ `/* always accept better solutions */`
  else
    $p \leftarrow \exp(-T^{-1} \cdot \frac{E'-E}{E})$ `/* acceptance probability formula */`
    $a \leftarrow 1$ with probability p, 0 otherwise
  if $a = 1$ **then**
    $(R, E) \leftarrow (R', E')$ `/* accept solution` R' `*/`
  $T \leftarrow (1 - \tau) \cdot T$ `/* lower the temperature */`
until R *is well enough*

to be deterministic and efficient. Depending on the concrete problem, describing solutions indirectly by a representation can have the advantage that a mutation can be defined more easily.

An *evaluation function* computes the energy of a solution candidate given its representation. This may be split into two separate steps: decoding a solution from the representation and calculating the energy of the resulting solution.

Two randomized functions in the Simulated Annealing algorithm define the search space exploration. An initial solution representation is created by the *initialization function*. Starting from there, a *mutation function* is applied repeatedly: it forms a new solution representation by changing the current one only slightly. For any input, the set of all possible outputs of this function defines the *neighborhood* of the given representation.

The initialization function does not necessarily need to be randomized. However, if it is not, or if it uses a clever heuristic to start with an initial "guess" of a good solution, it is still important to design the mutation function such that it is still possible to escape from the local optimum of the initial solution quite easily. Otherwise, the simulation might stay at the local optimum of the initial solution, instead of exploring solutions near a different local optimum as well. Ideally, the initial solution is irrelevant for the quality of the result, and at the very most only an optimization of the running time.

The ability to escape any local optimum is mainly affected by the design of the mutation function. It defines a *search graph*, in which representations are nodes and neighbors are connected by edges. In order to make every representation reachable, starting from anywhere in this graph, the search graph needs to be connected. Also, the number of mutations required to make all representations reachable should be kept low. This can be achieved by designing the mutation function such that it sometimes changes the representation only slightly, and sometimes significantly, chosen randomly.

The Temperature Curve

The main idea of Simulated Annealing is to define a *temperature parameter* T, which is lowered over time, to simulate a natural cooling process. This parameter influences the

probability of accepting a worse solution intermediately, hoping that a better solution is found later — a process which is required to escape from local optima.

Typical Simulated Annealing implementations resemble a natural temperature curve starting with a fixed temperature T_0. At any point in time, the temperature change is proportional to the difference of the temperatures of the object and the environment. This results in a hyperbolic temperature curve. However, how fast the object is cooling down is determined by the proportionality factor, basically the thermal conductivity and capacity of the object. Analogously, we introduce the parameter τ which determines the temperature delta $\tau \cdot T$ for every iteration and thus how fast the simulation is cooled down (see line 14 in Algorithm 3.1).

The problem with such a curve is that its parameters T_0 and τ have to be tuned depending on each instance to give great results: very large instances require a slower cooling procedure, since the solution space is much larger and more complicated compared to that of smaller instances. However, when the total running time of the algorithm is to be fixed, the parameter should be higher for larger instances. This is because each iteration takes more time in this case, reducing the total number of iterations the algorithm is running. That means that to reach the same final temperature, the parameter τ needs to be larger. In the experiments in Chapter 6, we analyze this in detail.

The Acceptance Probability Formula

In a Simulated Annealing algorithm, by applying the mutation function we obtain a neighbor solution which is then evaluated. The energy of this new candidate is compared to the current solution: if the energy was decreased, i.e. a better solution was found, it is definitely accepted as the new solution for the next iteration. Otherwise, the probability of accepting the candidate is

$$P_{\text{abs}} = \exp(-T^{-1} \cdot \Delta E)$$

where $\Delta E = E' - E$ is the energy difference, which is positive when the new solution is worse. We call this the *absolute* probability formula.

The problem with the definition from above is that the magnitude of ΔE depends on the magnitude of E, which in turn highly depends on the problem instance. It would be difficult to define a useful value for T_0 leading to acceptance probabilities suitable for the concrete instance. To handle this, we *normalize* the energy difference by dividing it by the energy of the current solution. We call this the *normalized* probability formula:

$$P_{\text{norm}} = \exp(-T^{-1} \cdot \frac{\Delta E}{E})$$

This is the formula we use in our algorithm (line 10).

4. Optimization Problems in Wind Farms

In this chapter, we describe the problems we attempt to solve in the rest of this work. First, we formally describe a cabling problem for a wind farm. We consider the special case in which the farm has only a single substation. The cabling problem assumes substations with already fixed locations. Since the locations have a huge impact on the cabling costs, we also formulate the problem of finding their optimal positions.

4.1 The Cable Layout Problem

As discussed in the introductory chapter, to minimize the construction costs of a wind farm, we compute an optimal cable network connecting the turbines. For now, we assume fixed locations of turbines and substations, and only consider the cabling up to the substations, ignoring how the power is delivered from the substations to the electrical grid. That means, the cables we consider connect either a pair of turbines to each other or a turbine to a substation.

For each such possible connection, we can choose between different types of cables, differing in their cost and the maximum power rating they are designed for. Specifying a maximum power rating for cables is necessary because of the energy loss inside the cable due to a non-zero resistance: the electrical energy is not really lost but rather converted into thermal energy. This heat has to dissipate properly in order to avoid over-heating and in turn damage to the cable's insulation. The maximum power rating is thus a result of the thermal limit of the cable's design (and its environment, to be precise), which we call the *cable capacity*[1]. Also, for each substation, a maximum total power rating of all connected turbines is specified, which we call the *substation capacity*. The *Wind Farm Cable Layout Problem* aims to choose the cable types such that the overall cabling costs are minimized, and the capacity constraints for the cables and substations are satisfied.

Before describing the whole problem in a more formal way, we need a notation for the input data, that is the power ratings of turbines and capacities, as well as the costs of the cables.

Power Ratings

In a wind farm, power ratings and capacities come into play on several occasions. Turbines generate a particular amount of power. Likewise, cables, substations and other electronic

[1] Not to be confused with the capacity in electrical physics.

components have a rating limit they are designed for, where components with higher limits usually cost more.

The power a turbine generates depends on multiple factors. It is self-explanatory that the wind strength plays the most significant role, but it affects all turbines in the same wind farm more or less equally. In practice, the ratings of all turbines in the same wind farm are very similar to each other, since they are usually of the same type and their exact location affects the wind strength only slightly. However, turbines have a specified maximum power rating which they never exceed, even if they could produce more. This quantity is called the turbine's *nominal power*, and is generally used to dimension the cabling and other electrical components for power transportation.

Therefore, instead of expressing power ratings in units of watts, we simplify the notation a lot by expressing them in multiples of the nominal rating of one turbine. Furthermore, we only consider the situation in which every turbine produces the nominal power, that means its production is set to 1. The power limits of cables and substations are rounded down to whole numbers.

Cabling Costs

The cost to lay a cable from one node to another is basically composed of three parts: the connection points, the cable itself, and the cost of installing the cable between the nodes. The costs for the connection points in both nodes are independent of the length of the cable, but depend on the power rating routed through. The cost of the cable itself is the product of the cable length and the cost per length of the cable type used. Which cable type suites the connection mainly depends on the power rating. Finally, the cost to install the cable is significant especially when the cable is to be run underground, in which case it mainly depends on the terrain between the nodes.

When solving the cabling problem in this work, we are not interested in how the cabling cost is composed in detail. Not even the length of the cable is relevant on its own. Instead, for each edge and available cable type its total cost and the maximum power rating are interesting. These values can be precomputed for every edge separately. Figure 4.1 shows this data for an edge for which four cable types are available. For example, the first cable supports the power of two turbines. This edge does not support the power of more than 9 turbines.

For any edge, we call a cable *superseded*, if there is any other cable available which has both a lower or the same cost as well as a greater or the same capacity. Superseded cables are not considered at all in our algorithms, since there is always a better (or equally good) option. This has the consequence that when ordering the cables by their cost, they are also automatically ordered by their capacities, and vice versa. This order implies the *cable number*.

Note that our cable layout problem allows only one cable to be placed on each edge. However, if the user wants to allow a parallel wiring of multiple cables on the same edge, we can simulate this scenario by adding a virtual cable to the list of available cable types for that edge, by precomputing the cost and capacity of the wiring. Even more complicated setups can be modeled in an analogous way.

The Input Data

Formally, an instance of the Wind Farm Cable Layout Problem has the following input data. The wind farm has t turbines and s substations. We are given the set of possible connections, and for each of them a (separate) set of cable types, each with a capacity

Cable number	Capacity	Cost
1	2	1.3
2	4	1.5
3	7	2.0
4	9	2.2

Figure 4.1: An exemplary edge for which four cable types are available.

and cost. Furthermore, for each substation its capacity is specified, limiting the number of turbines which can be connected to it.

A cable network consists of the chosen cable type (maybe none) for every possible connection. It is called *valid* if it respects the constraints given by the capacities of the chosen cables and the substations. An instance is called *feasible* if a valid cable network for it exists. In this case, its solution is a valid cable network with minimal total costs of the chosen cables.

Using the following mathematical model, this problem can be defined using a flow network, which is then formulated as a mixed integer linear program.

The Wind Farm as a Graph

We model the wind farm as a simple directed graph $G = (V, E)$ in which turbines and substations are nodes. We denote them as separate sets: V_T is the set of turbine nodes, and V_S are the substation nodes, such that together they form the set of all nodes $V = V_T \cup V_S$. Each pair of nodes, which can be connected by a cable, is modeled as an edge in this graph. The direction of the edge is arbitrarily chosen, yet it needs to be consistent to have a notation of direction when talking about a positive or negative flow on an edge. The set of edges is denoted by the symbol E.

For each edge, we are given a set of cable types, each with a capacity and a cost. To notate these numbers, let $k \in \mathbb{N}$ be the maximum number of cable types available on any edge. For any edge, for which less than k cable types are available in the original input, add dummy cable types with zero capacity (and arbitrary cost) such that every edge has k cable types. The cable types are ordered by ascending capacity. Then, for edge (v, w) and $i \in \mathbb{N}_{\leq k}$, the capacity of the i-th cable is denoted as $u_{vwi} \in \mathbb{R}_{\geq 0}$ and the cost as $c_{vwi} \in \mathbb{R}_{\geq 0}$. To simplify the notation in the special case of an edge with no power routed through, we define these variables also with a cable type index of 0: $u_{vw0} := 0$ and $c_{vw0} := 0$.

Finally, the capacity of substation $v \in V_S$ is denoted as $m_v \in \mathbb{N}$.

The Flow Network

Before modeling the optimization problem as a minimum-cost flow problem as explained in Section 3.1, we need to define a modified version of the above graph. Our cabling problem defines substation capacities which we have to model somehow in the flow problem. Recall that an instance of the minimum-cost flow problem contains the balance function $b : V \to \mathbb{R}$, which is negative for sinks, where the absolute value models the amount of consumed flow at that node.

However, it can not be used for a flexible amount of consumption, which is what we need to model for our substations: their received amount of power is not fixed, yet limited by their individual capacities. Since the sum of substation capacities might be greater than the total amount of power produced in the wind farm, we have to deal with this flexibility in the assignment of power to the substation nodes. But since our definition of a flow network

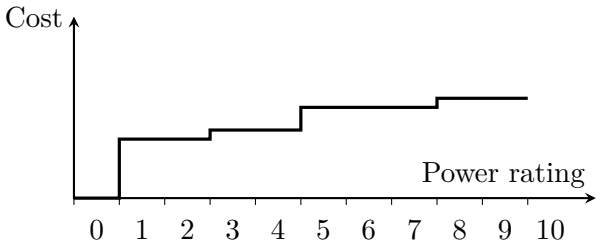


Figure 4.2: The edge cost function corresponding to the cable types from Figure 4.1.

uses fixed values for the consumed amount at a sink, we have to model the substation capacities using a workaround. For this, we introduce a *dummy sink* σ. It serves as the global sink in the flow network, i.e. it is the only sink with a balance of $-t$. We connect it with dummy edges to the actual substation nodes. The edge does not charge any cost but has the substation capacity as its capacity.

Formally, we define $G' = (V', E')$ as our modified version of $G = (V, E)$: The set of nodes is $V' := V \cup \{\sigma\}$. For the edges, we first define the set of dummy edges connecting the substations to our global sink as $E_\sigma := V_S \times \{\sigma\}$. Together with the original edges, they form the edge set $E' := E \cup E_\sigma$.

Now we are ready to model the optimization problem as the minimum-cost flow problem $(G', u, b, c_{(\cdot)})$. Note that since we notate power ratings as multiples of the nominal power of a single turbine, our flow values are whole numbers, and the edge cost functions are of the form $\mathbb{Z} \to \mathbb{R}_{\geq 0}$.

As explained above, the dummy edges are used to model the substation capacities: for substation $i \in V_S$, the edge $e = (i, \sigma) \in E_\sigma$ has capacity $u(e) := m_i$ and does not cost anything by setting $c_e(x) := 0$ for all $x \in \mathbb{Z}$.

For all other edges $e = (v, w) \in E$, we also have to define the capacity $u(e)$ as well as the cost function $c_e : \mathbb{Z} \to \mathbb{R}_{\geq 0}$, which are used to model the cable type parameters. The edge capacity is set to the maximum capacity among the available cable types, that is $u(e) := \max_{i \in \mathbb{N}_{\leq k}} u_{vwi}$. For a given flow value $f(e)$, we set $c_e(f(e)) := c_{vwi}$ where i is the minimum value in $\{0\} \cup \mathbb{N}_{\leq k}$ such that $|f(e)| \leq u_{vwi}$ (i.e. the cost of the cheapest suitable cable). Since the flow absolutely never exceeds its upper bound $u(e)$, for $x \in \mathbb{Z}$ with $|x| > u(e)$, the value of $c_e(x)$ does not have to be defined. Consequently, a suitable i is always found.

The resulting edge cost functions are all of the same basic shape: a step-wise constant function with one segment for each available cable type. Figure 4.1 shows the cost function for the cables from Figure 4.2. As expected, it does not charge any cost for a power rating of zero, since in this case no cable is laid at all. The value of the edge cost function for any value greater than the maximum capacity among the available cable types is not relevant, as the edge's capacity will constraint the flow value.

Solving the minimum-cost flow problem $(G', u, b, c_{(\cdot)})$ results in the cost-optimal flow $f^\star$, in case it is feasible. For any edge $e \in E$ (the original edges), the flow value $f^\star(e)$ denotes the power routed through a cable on that edge. If the flow problem is infeasible, our original cabling problem instance is also infeasible.

From a feasible flow $f^\star$, a cable layout can be computed trivially. A cable type for each edge is determined by choosing the cheapest cable with capacity at least the absolute edge's flow value. If there is no flow on an edge, no cable is chosen.

Mixed Integer Linear Programming

This network flow problem can be easily expressed as a mixed integer linear program (MILP) with integer variables and real coefficients. Note that the following model computes the choice of cable types directly, and the flow values only appear as helper variables in the program, as they are not important for a solution to the original problem. Also, we do not need the dummy edges introduced for the flow network. Instead, we use the original graph $G = (V, E)$.

To model the choice of cable types on each edge $(v, w) \in E$, we introduce the set of binary variables x_{vwi} for each cable type. If $x_{vwi} = 1$, the i-th cable type was chosen for edge (v, w). For every edge, at most one such variable can be 1.

Furthermore, we introduce the integer helper variable f_{vw} for each edge $(v, w) \in E$, which corresponds to the flow value $f(v, w)$ in the flow network. It is not required for the result, but to model the constraints in the flow network: the capacity constraint for each edge and the flow conservation at each node. To simplify notation, we alias $f_{wv} := -f_{vw}$ for any edge $(v, w) \in E$. Note that this does not add more variables to the linear program.

Usually, the edge cost in a flow network depends on the flow through that edge. However, since in our case the cabling cost only depends on the choice of cable, which is modeled by the separate set of variables x_{vwi}, it can be expressed without the flow helper variables f_{vw} in our linear program. Before listing all equations and constraints of the program, we first show how the cost and capacity of an edge (v, w) can be expressed as sums of products using the variables above.

$$\text{Cost of edge } (v,w): \quad \sum_{i \in \mathbb{N}_{\leq k}} c_{vwi} \cdot x_{vwi}$$

$$\text{Capacity of edge } (v,w): \quad \sum_{i \in \mathbb{N}_{\leq k}} u_{vwi} \cdot x_{vwi}$$

Similarly, the excess of power at node v can be expressed as a sum of flow values of the incident edges.

$$\text{Excess at node } v: \quad \sum_{w \in N(v)} f_{vw}$$

The objective function, the total cabling cost, is then a sum of the cost over all edges.

$$\text{Total cabling cost:} \quad \sum_{(v,w) \in E} \left(\sum_{i \in \mathbb{N}_{\leq k}} c_{vwi} \cdot x_{vwi} \right)$$

This term is to be minimized under the following constraints:

$$x_{vwi} \in \{0, 1\} \qquad \forall (v, w) \in E, i \in \mathbb{N}_{\leq k} \tag{4.1}$$

$$f_{vw} \in \mathbb{Z}_{\geq 0} \qquad \forall (v, w) \in E \tag{4.2}$$

$$\sum_{i \in \mathbb{N}_{\leq k}} x_{vwi} \leq 1 \qquad \forall (v, w) \in E \tag{4.3}$$

$$f_{vw} \leq \sum_{i \in \mathbb{N}_{\leq k}} u_{vwi} \cdot x_{vwi} \qquad \forall (v, w) \in E \tag{4.4}$$

$$\sum_{w \in N(v)} f_{vw} \leq m_v \qquad \forall v \in V_S \tag{4.5}$$

$$\sum_{w \in N(v)} f_{vw} = -1 \qquad \forall v \in V_T \tag{4.6}$$

The first two conditions (4.1), (4.2) define the range for the variables in the linear program, and condition 4.3) ensures that only one cable type is chosen for each edge. Condition (4.4) constraints the power throughput of each edge to the capacity of the chosen cable type. This models the edge capacity constraint (3.2) from our definition of the flow problem in Section 3.1.

Furthermore, condition (4.5) constraints the total power coming in at each substation to its capacity. The balance is set to -1 for each turbine by condition (4.6). Together, these constraints model the flow conservation constraint (3.3) from our definition of the flow problem.

This mixed integer linear program has $\mathcal{O}(|E| \cdot k)$ variables and $\mathcal{O}(|E|)$ constraints. For complete graphs as the input, i.e. when $|E| = |V|^2$, its size is quadratic in the number of turbines.

The Substation Cable Layout Problem

We also consider the special case with only a single substation, that is $s = 1$, which we call *Substation Cable Layout Problem*. Here, it isn't necessary to specify the substation capacity, since there are just two possible cases: it can handle the power of all turbines, then the capacity is always respected; or it can not handle it, but then a feasible solution does not exist.

As we will see later in our experiments in Chapter 6, this special case is much simpler to solve.

4.2 The Substation Assignment Problem

The Wind Farm Cable Layout Problem can be split into two layers. First, for every turbine $v \in V_T$, we find a substation $w \in V_S$, to which its power is routed. We say, v is *assigned to* w. We call this sub-problem the *Substation Assignment Problem*. Then, for every substation $w \in V_S$, the Substation Cable Layout Problem is solved in which only the substation w and the turbines assigned to w are considered.

As its input, the Substation Assignment Problem gets the exact same information as the Wind Farm Cable Layout Problem. The difference is that, instead of finding the optimal cables for each possible interconnection, its output is a function $q : V_T \to V_S$ which assigns turbine $v \in V_T$ to substation $q(v) \in V_S$ leading to minimal cabling cost when computing the cable layouts for each substation separately. The Substation Assignment Problem is an optimization problem, so we define an objective function which an optimal solution $q^\star$ minimizes.

For this, we have to describe how the input for each individual Substation Cable Layout Problem is constructed, based on an assignment function q. For a given substation $w \in V_S$, we set $V_w := \{v \in V_T : q(v) = w\}$, i.e. the turbines of which the power is to be routed to substation w. However, the paths for routing the power of these turbines may not necessarily only use these nodes. Instead, the power of turbine $v \in V_w$ might use turbines from $V_T \setminus V_w$ as intermediate nodes, i.e. nodes of which the power is routed to a substation other than w.

We observed that this is the case for most representations. Why this happens and how this can be handled is explained in Chapter 5, when we describe an algorithm implementing the two-level approach.

4.3 The Substation Positioning Problem

So far, we discussed the problem of finding cost-optimal cable layouts in wind farms where turbines and substations have predefined locations. However, when only the locations of the turbines are fixed, but the substations can be positioned freely or with some degree of freedom, the cabling costs can be reduced further.

For this, we introduce the *Substation Positioning Problem*: Given t turbines and s substations, find the optimal locations of the substations such that the total cabling costs are minimized. Now, we also need to consider the costs for the export cables, as they depend on the choice of the substation locations – previously they where not relevant as the locations had been fixed. That means, among all possible combinations of substation locations, find the one resulting in the lowest cost as found by the Wind Farm Cable Layout Problem plus the cost for the export cables.

For simplicity reasons, we no longer define different capacities for the substations. We are rather given a single substation capacity $m \in \mathbb{N}$, which is the capacity of all substations in the wind farm.

Before we model the problem formally, we have to discuss which locations are considered for the substations. In this work we use discrete locations, that is we are given a set L of possible substation locations, among which the optimal ones are to be chosen. Also, how are the costs for export cables, and the cost for the cables connecting turbines to the substations (which have no longer a fixed length) given in the input data?

For this, we define our node set as $V := V_T \cup V_L$ in which V_T are again the turbine nodes, and V_L are the nodes representing all possible substation locations. The set of edges are then $E := (V_T \times V_T) \cup (V_T \times V_L)$. Together with the nodes, we have the graph $G := (V, E)$. This enables us to model the cabling cost as precomuted tabular data similar as before: for every cable from a turbine to a (possible location of a) substation, we have a set of cable types each with cost and capacity, like in our Cable Layout Problem. We also need to model the cost for an export cable from $v \in V_T$ to the grid. For this, we assume that for any possible substation location, we already know the nearest grid point, or to be more precise, the one for which an export cable is as cheap as possible. This cost is then given as $c_v \in \mathbb{R}_{\geq 0}$.

Mixed Integer Linear Programming

There is no straight forward transformation of this problem into a flow network problem: we would need to restrict the number of location nodes V_L used to s, the number of substations.

On the other hand, the problem can modeled as an MILP similar to our model of the Cable Layout Problem. For this, we add the set of binary variables y_v for $v \in V_L$, which are 1 if a substation is placed at the location associated with node v. Their sum must not exceed s. Also, the flow on an edge from a turbine $v \in V_T$ to a substation location node $w \in V_L$ has to be zero if there is no substation, that is $y_w = 0$. This is expressed by the disjunction:

$$\sum_{v \in V_T} f_{vw} = 0 \ \vee \ y_w = 1$$

To add this condition to our MILP, we need to reformulate it as a linear expression. For this, observe that although the excess of a location node is not fixed, like for regular nodes which have a balance, it never exceeds m due to the capacity constraint of a substation:

$$\sum_{v \in V_T} f_{vw} \leq m$$

Also observe that $(y_w = 1) \Leftrightarrow (1 - y_w = 0)$, which lets us construct the linear expression as

$$\sum_{v \in V_T} f_{vw} \leq m \cdot (1 - y_w)$$

In the case the substation is used, the right-hand side evaluates to m, such that the expression constrains the incoming flow of the substation to m. In the other case, the right-hand side evaluates to 0 such that no incoming flow is allowed.

We construct the full linear program by reusing the constraints (4.1)-(4.4) and (4.6) from above, where V_S is substituted with V_L. Note that we do not need (4.5), since the above expression already models the substation constraints. We add the following additional constraints:

$$y_v \in \{0, 1\} \qquad \forall v \in V_L \tag{4.7}$$

$$\sum_{v \in V_L} y_v \leq s \tag{4.8}$$

$$\sum_{v \in V_T} f_{vw} \leq m \cdot (1 - y_w) \qquad \forall w \in V_L \tag{4.9}$$

So far, we did not consider the cost for the export cables. We simply add them to the objective function: when placing a substation at the location node $v \in V_L$, that is $y_v = 1$, the cost c_v is charged. The objective function to be minimized then becomes

$$\sum_{(v,w) \in E} \left(\sum_{i \in \mathbb{N}_{\leq k}} c_{vwi} \cdot x_{vwi} \right) \quad + \quad \sum_{v \in V_L} c_v \cdot y_v \,.$$

5. Heuristics for Wind Farm Problems

In Section 4.1, we introduced the Wind Farm Cable Layout Problem. We proposed an MILP which we implemented using a generic solver for our experiments. However, in this chapter we propose an alternative algorithm solving the layout problem heuristically. The first approach is based on Simulated Annealing, which we describe in the first section. The algorithm is then improved in the following sections.

5.1 An Algorithm based on Simulated Annealing

Our first algorithm which heuristically solves the *Wind Farm Cable Layout Problem* efficiently is based on Simulated Annealing, a randomized search space exploration heuristic introduced in Section 3.2. The algorithm uses an indirect representation of a solution candidate, which is altered over time using a mutation scheme.

This section is structured as follows. First, the algorithm is described in details. We then evaluate it for randomly generated instances. The strengths and limits of the algorithm are discussed, before we try to fix this in the next section by an improved version.

The introduction of Simulated Annealing in Section 3.2 mentions the "building blocks", which need to be defined to construct a complete heuristic for a concrete problem. This subsection presents the representation of the cabling and functions needed for initialization and mutation of this representation. Evaluating it leads to the cabling cost, which is to be minimized by the Simulated Annealing heuristic.

The Representation

Recall that the representation shall describe a solution candidate indirectly. A solution to the Wind Farm Cable Layout Problem assigns each edge in the input graph a cable type.

Our representation is composed of two structures: a *potential field* and *edge cuts*. The potential field[1] $p : V \to \mathbb{N}_{\leq n}$ assigns each node $v \in V$ in the graph a number $p(v)$, its potential. Recall that $\mathbb{N}_{\leq n} = \{1, \ldots, n\}$, where n is the number of nodes. Each node has a different value, so p is a permutation of the node indices. Its concrete role becomes clear when the evaluation function is explained below. The edge cuts $H \subset E$ are the edges which shall not be used for cabling. Together, they form the representation as a tuple $R = (p, H)$.

[1] not to be confused with the potential field in electrical physics

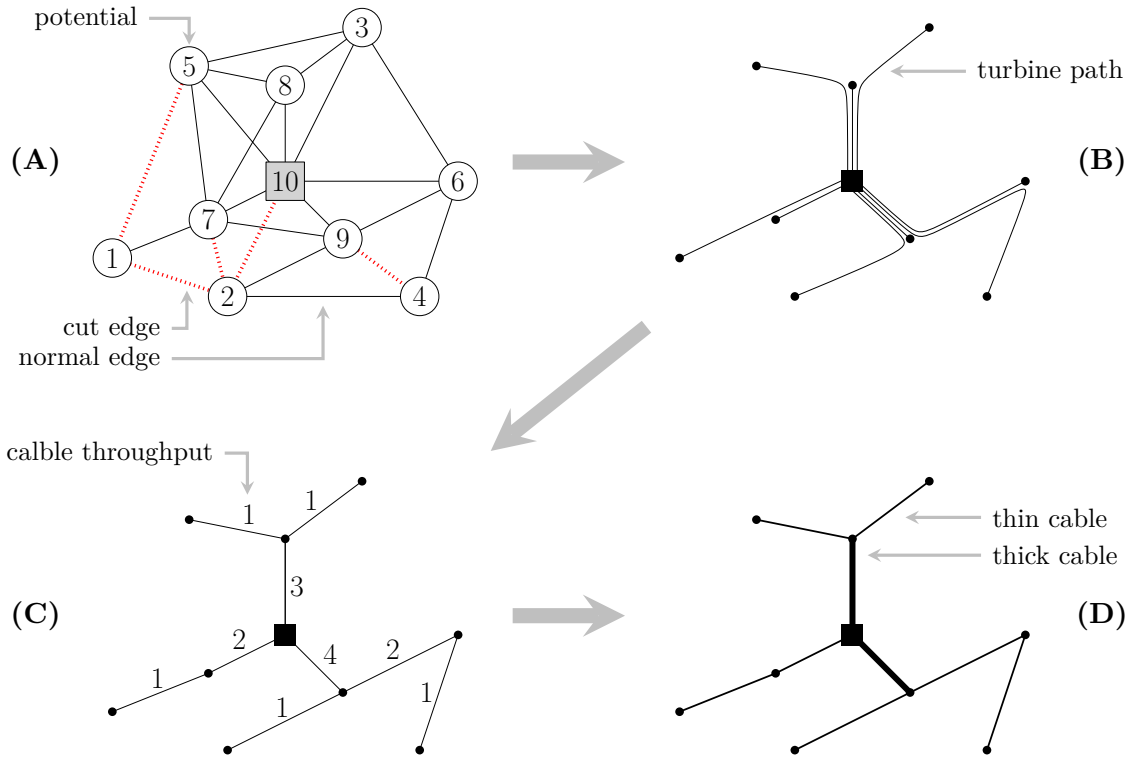


Figure 5.1: An example encoded in our representation (A) as well as the intermediate representations used during the evaluation: the turbine path representation (B), the cable throughput representation (C), and finally the cable types (D).

We create a representation of the initial solution as follows. For each node $v \in V$, we find its nearest substation. We order the nodes by the distance to their nearest substation in descending order. The initial potential $p(v)$ of each node is set to its rank in this ordered list. This results in substations having the largest potential values, and the turbines most far away from any substation have the lowest numbers. The initial edge cut list H is empty.

The Evaluation Function

Evaluating a representation R requires decoding it into a solution, here the choices of cable types for each edge. Then, the cost of the solution is merely a task of summing up the cable costs, which is trivial in this case.

To simplify the explanation of how a representation, given the potential field and edge cuts, is decoded into cable type choices, we introduce two intermediate representations: the *turbine path representation* and the *cable throughput representation*. Their correlation is visualized in Figure 5.1. In the following, we explain these representations and how to decode them in reverse order: from the last to the first.

The cable throughput representation (C) specifies for each edge the direction and amount of power routed through. From this, it is trivial to determine the corresponding cable types (D) by choosing the most inexpensive cable still supporting the requested throughput. Summing up all cable costs finally leads to the total cost of the representation. However, if for some edge there is no cable available which supports the requested throughput, we charge the cost of the thickest cable plus a penalty cost. We chose a penalty proportional to the square of the amount of power the cable is exceeded, as well as inversely proportional to the temperature. This means, penalties increase over time. Similarly, we enforce the substation capacity constraints by charging a penalty proportional to the amount of power

it is exceeded by the solution. The proportionality factors for both types of penalties are tunable parameters of the algorithm.

The turbine path representation (B) specifies for each turbine, where its produced power travels through the network, finally reaching a substation. Every turbine has its own path encoded in this intermediate representation. From these paths, the cable throughput representation (C) is computed as follows. Given the paths of all turbines, we find the cable throughput for an edge by counting how many turbine paths use that edge. Recall that we have a notation of edge direction, which we use here to eliminate positive and negative flow on an edge: for every path which uses an edge in its direction, we increase a counter for that edge; whenever the same edge is used in its opposite direction, we decrease the counter. Finally, the absolute value of the counter is the throughput a cable on that edge needs to support.

The actual representation used in the algorithm (A), as explained above, contains the potential field p and the edge cuts H. Using that, Algorithm 5.1 computes a path π for every turbine separately, which we explain in detail. Starting from the turbine v, the path is created incrementally node by node. The process ends when a substation was reached (line 4). At each step, a restricted neighborhood $N'(x)$ of the current node x is considered, from which we select the node maximizing some expression (line 8). This expression is based on the idea of a "slope" in a height field, but with some adjustments, which we explain in detail below. The restricted neighborhood, from which the best node is chosen, are the neighbors of x of which the edge is not in the edge cut set and which is not already part of the path (line 5). When the restricted neighborhood is empty (line 6), the path can not be terminated at a substation, and the whole representation is considered invalid. This is handled by setting its energy to ∞[2].

The formula used in line 8 needs some more explanation: From the restricted neighborhood, nodes with higher potentials should be preferred (such that the potential field serves as a "guide" for the flow direction), while at the same time near neighbors (i.e. short cables) should be preferred. The most intuitive formula would be $(p(w) - p(x)) \cdot {d_{xw}}^{-1}$, i.e. basically the slope in the potential field in the edge direction, which we originally used in our algorithm. The problem here is that when no node with higher potential exists in the restricted neighborhood, this expression is always negative. This in turn means that longer distances are preferred, which we want to avoid. We divide the potential difference by n, i.e. normalize it to the range $(-1, 1)$, and then add 1 to that result. This gives a positive value in the range $(0, 2)$ for every potential difference, while satisfying our original intent: neighbors with higher potential as well as smaller distance are preferred.

The Mutation Function

When mutating a solution, we randomly choose one of the following different kinds of operations. Based on the representation $R = (p, H)$, they form a new representation $R' = (p', H')$.

The first method swaps two random values in the potential field p (and keeps the edge cuts $H' := H$). We choose two distinct nodes $a, b \in V$ randomly and then set $p'(a) := p(b)$, $p'(b) := p(a)$.

The second method also modifies the potential field p, but tries to only modify the value of one node notably. Since the potential field is defined as a permutation of node indices, we can not just change one entry. Instead, we adjust the potentials of other nodes very slightly

[2] We found that this case rarely occurs (less than one in a thousand cases). Whenever the energy of a representation evaluates to ∞, it is always discarded by Simulated Annealing. Effectively, this means that a new representation is chosen by mutating the current one again.

Algorithm 5.1: Constructing a turbine path from the representation

Input: Graph $G = (V, E)$, Node distances d_{vw}, Capacities u_{vwi}, Costs u_{uvi}, Representation $R = (p, H)$, Turbine v
Output: Path π of nodes from v to a substation

$\pi \leftarrow$ new list of nodes
$x \leftarrow v$
append x to π
while x *is not a substation* **do**
  $N'(x) \leftarrow \{w \in V \mid (x, w) \in E \setminus H,\ w \notin \pi\}$
  if $N'(x) = \emptyset$ **then**
    return infeasible
  $x \leftarrow \arg\max_{w \in N'(x)} \left((1 + \frac{p(w) - p(x)}{n}) \cdot {d_{xw}}^{-1} \right)$
  append x to π
return π

such that the result is again a valid permutation. For this, we also choose two distinct nodes $a, b \in V$ randomly, where a is the node of which the potential is to be changed significantly: it is set to the potential of b: $p'(a) := p(b)$. For now, assume that $p(a) < p(b)$, i.e. the potential of a has been raised. We now decrease the potentials of all nodes with a potential in between $p(a)$ and $p(b)$ by 1. This results in a permutation again, since we just "rotated" the potential values of some node subset. Analogously, if $p(a) > p(b)$, the potential of the nodes with potentials between $p(b)$ and $p(a)$ are increased by 1.

The third method modifies the edge cuts H while keeping the potential field $p' := p$ as is: we add or remove a random edge to the set of edge cuts. The probability to add a new edge cut (instead of removing one) depends on the current cardinality of H (in order to probabilistically level out the number of cuts to some predefined number, for example $\mathcal{O}(\sqrt{n})$ with a tunable factor).

5.2 Improving the Algorithm

The algorithm as seen so far does not perform as well as we hoped at first. We did early experiments in which we noticed several problems of this basic algorithm. Therefore, we implemented some improvements which we discuss in this section.

Dynamic Temperature Curve

The temperature curve in the original Simulated Annealing heuristic is fixed over time (measured in the number of iterations). However, we observed that for larger instances the algorithm needs more iterations to reach a good solution compared to smaller instances, since more mutations need to be applied, as every mutation only performs one local change. For this, the temperature curve should be adjusted depending on the instance. We came up with two possible solutions: making T_0 and τ depend on the instance size n, or adjusting the temperature delta dynamically during runtime such that the curve is flattened as long as a good solution has not yet been reached.

As the first approach is just a matter of adjusting the algorithm's parameters before running it, we implemented the second idea into our algorithm. For this, we introduce a new parameter which is computed during running time: the *activity* μ measures how volatile the current search exploration behaves. This is done by exponentially smoothing the

probability of accepting worse solutions (whether they are accepted or not), by computing the following formula after line 10 in Algorithm 3.1:

$$\mu' := \alpha_{\text{smooth}}\, p + (1 - \alpha_{\text{smooth}})\, \mu$$

The value of μ is initialized with 1. The parameter α_{smooth} is a tuning parameter which defines how easily the activity is influenced by temporary fluctuations of the probability value. We set it to 10^{-3} in our implementation. Adjusting the temperature after each iteration then becomes the formula

$$T' := (1 - \mu\tau) \cdot T$$

This approach follows the intention that in the early stage of the cooling simulation, we basically explore the search space globally. Not much time should be spend in this phase. It is important that enough time is available for the later stage, in which the temperature should be rather cool such that the already quite good solutions can be further improved. This could be done by simply raising the value of τ. However, the temperature in that late stage should not be *too* low, as that would contradict to the idea of Simulated Annealing: that a worse solution should be accepted temporarily.

We want a temperature curve which falls quickly in the beginning, but slower towards the end. This is achieved by the formula for our dynamic temperature curve as we expect a decreasing activity during the runtime of a computation: making the temperature change proportional to the activity deforms the otherwise exponentially decreasing curve to a sub-exponential shape. Figure 5.2 shows both the standard and dynamic temperature curves for one of our experiments. Note that both curves start and end with (approximately) the same temperature levels. Compared to the standard curve, the dynamic one thus basically *compresses* the early phase, while *expanding* the late stage of the computation. In Section 6.2 we analyze experimentally how that influences the quality of the solution.

Maintaining Multiple Instances

So far, our algorithm only handles one solution at a time. It is mutated and evaluated iteratively, resulting in a single search space exploration. The main difficulty with such an iterative exploration is escaping a local (non-global) optimum, when a worse solution needs to be accepted temporarily. Although Simulated Annealing is designed to make this possible, our experiments showed that the search exploration gets caught in a different local optimum every time the algorithm is ran with a different random seed. This is shown in detail in Chapter 6.

This led us to the idea to run multiple instances of the same algorithm in parallel, each with a different random seed. They explore the search space independently, and when one instance gets caught in a local optimum, others are not affected by that. We call the instances of the Simulated Annealing algorithm *threads*[3]. Note that every thread maintains its very own temperature, and they do not exchange any information.

Furthermore, we had the following additional idea for managing the computation threads. Initially, a configurable number of threads are started. From each thread, the currently best solution, its energy and the activity can be queried. A meta algorithm decides when a thread is likely to be caught in a local optimum, in which case its computation is stopped. We propose to use the activity as a measure of the likelihood that continuing a computation does not make much more sense. For this, an activity threshold value μ_{thresh} is introduced as a configurable parameter. As soon as the activity falls below this value, the thread is

[3]Not to be confused with the term in parallel computing.

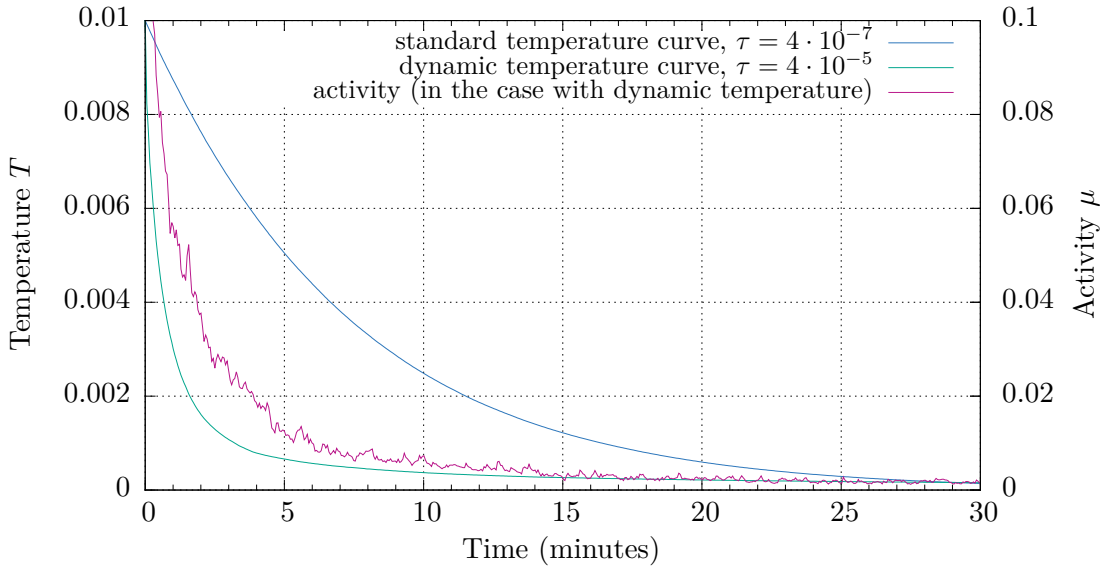


Figure 5.2: The standard and dynamic temperature curve of one of our experimental instances, as well as the activity corresponding to the computation with the dynamic temperature curve. Both temperature curves start at $T_0 := 0.01$. The parameter τ is chosen separately in the two cases such that both curves reach approximately the same final temperature of about 0.00015 (= 1.5 % of T_0).

considered *dead*, in which case it is stopped. This saves computation power now available either for the other threads, or for a whole new thread.

The solution of a stopped thread is not discarded, as the meta algorithm regularly queries all threads for their best solution seen so far. We globally track the currently best global solution, which is then the output of the overall algorithm. So only because a thread is considered dead, it does not mean that its solution is bad, it only means we do not give it more time to improve the computation.

A similar idea could be that instead of distributing the available computation power evenly among all threads, but rather proportional to their activity. This slows down the computation of inactive threads, while more effort is put into improving the solution of active threads.

Even more complex and smarter strategies could be developed for managing the threads. We leave this for future work.

Recovering from Bad Local Optima

We observed in our early experiments that the Simulated Annealing algorithm sometimes takes very long time to improve its best solution seen so far. Our assumption is that this means that it is then caught in a local optimum from which it is hard to escape again (we discuss this in detail in Chapter 6). Therefore, we propose to track the number of iterations since the last improvement over the best solution seen so far. When this number exceeds a configurable threshold value, we reset the computation to that best solution, starting the search space exploration again from there.

An alternative to this idea is to *branch* the thread in such cases: one thread continues the computation as according to the original algorithm, while a second one restarts from the best solution seen so far, increasing the number of computation threads by one.

We do not want the number of threads to raise unconditionally, since the available running time has to be divided among them, reducing the running time per thread. For this, we configure our algorithm with a maximum number of threads. When exceeding this value, one thread should be killed. Note that we want to allow the Simulated Annealing algorithm to perform a predefined number of minimum mutations on newly created threads, before we consider it *mature*. Only mature threads are allowed to be killed by the meta algorithm. From all mature threads we kill the worst, i.e. the one with highest energy of the best solution it has seen so far.

5.3 Towards Evolutionary Algorithm

In our experiments, we made the observation, that different threads explore the search space at very different local optima. The solutions all have similar energy levels, but propose very different cable layouts, even the turbines are often assigned to different substations when comparing any two solutions.

This led us to the following idea which extends the above meta algorithm with a new event: we *cross* two solutions, that is, we create a whole new solution based on information from the both. This approach is similar to how biological evolution works, and is known as an *Evolutionary Algorithm*, in which our representation can be thought of as a genetic code.

We propose to integrate this technique in our existing algorithm as a separate phase. For this, the total running time is split into a longer first phase, in which the existing algorithm is run. Afterwards, the best pairs suitable for crossings are chosen, and their representation is crossed. The second, shorter phase is then used for solving these crossings.

We consider a pair of solutions suited for crossings if the two solutions have a good *compatibility* but high *diversity*. The compatibility measures how well we can construct a crossing from the both solutions, which hopefully results in an even better solution. Each crossing results in two new solutions.

A couple of questions arise, which have to be answered in order to complete the description of the algorithm: How is the pairwise compatibility and diversity of two solutions defined? How many pairs of solutions are considered for constructing crossings (i.e. how many crossings do we construct)? How is a crossing constructed? And finally, how much time of the total running time is used to solve these crossings? In the following, we provide suggestions to these questions.

Diversity and Compatibility of Two Solutions

We define the *diversity* of two solutions by the sum of the squared difference of the power ratings over all edges.

For the *compatibility* of two solutions A and B, given by representations $R_A = (p_A, H_A)$ and $R_B = (p_B, H_B)$, we decided to look at how their substation assignments differ. For this, we introduce the *substation assignment difference graph* $G_{\text{sad}} = (V_S, E_S, w)$, which is an edge-weighted undirected and complete graph over the substation nodes ($E_S := V_S \times V_S$). Its weight function $w : E_S \to \mathbb{Z}_{\geq 0}$ assigns each edge $e = \{a, b\} \in E_S$ the number of turbines which are assigned to substation a in one solution, and to substation b in the other. Then, we select the two substations $s, t \in V_S$ minimizing the value of the min-cut $(G_{\text{sad}}, s, t, w)$.

The min-cut $(G_{\text{sad}}, s, t, w)$ is a partitioning problem in which the optimal solution is a pair of node subsets $P_0 \subset V$ and $P_1 \subset V$ (with $P_0 \cap P_1 = V$) such that the sum of the weights of all cut edges is minimized. The *cut edges* are the edges of which one node is in P_0 while the other is in P_1. Several algorithms exist for the min-cut problem, such as the Stoer-Wagner algorithm [SW97].

Finally, the compatibility of the solutions is defined as the weight sum of the cut edges.

Constructing the Crossing

The partitions resulting from the cut are reused for constructing the crossing of two solutions; we call the two partitions P_0, P_1. To put it simply, we use the representation R_A of the first solution for regions which connect to substations to partition P_0 and for the rest of the farm, representation R_B is considered. However, we should not cut the representation data sharply at the border of the two regions, as this introduces discontinuities in the potential field. We rather interpolate between the two solutions in regions near the border.

For this, we define a *weight function* $z : V \to [0, 1]$, which assigns each node $v \in V$ (turbine or substation) a weight $z(v)$. Later, this weight determines the composition of the representations from the both solutions we are crossing: areas in which the node weight is 0 will use the representation R_A, while a value of 1 refers to R_B. However, since we use a real-valued weight function, an interpolation of the two solutions is possible, which we use for the boundary region in which we "cut" the two solutions.

We set $z(v) := 0$ if v is assigned to substation w in both solutions, and that substation is part of partition P_0 in the min-cut, that is $w \in P_0$. If however $w \in P_1$, we set $z(v) := 1$. In the case it is assigned to different substations in both solutions, we set $z(v) := 0.5$.

Finally, using z, we construct the potential field p' and the edge cuts H' of the new solution as follows. For all nodes $v \in V$, we linearly interpolate between p_A and p_B:

$$p_{A,B}(v) := (1 - z(v)) \cdot p_A(v) + z(v) \cdot p_B(v)$$

This can however not directly be used as the new potential field p', as it is real-valued and not a permutation of the node indices, as required by our definition of the representation. We construct a permutation by defining the value of $p'(v)$ as the rank of v in the list of nodes sorted by their value $p_{A,B}$.

The edge cut list H' is also constructed as a combination of H_A and H_B. For an edge $e = (v, w)$ we determine its related solution by looking at $z(v)$ and $z(w)$: if they are both 0, the edge will be part of H' if and only if it is part of H_A. Similarly, if both are 1, it is part of H' if and only if it is part of H_B. In all other cases, e is not in H'.

Now, we constructed a new solution $R' = (p', H')$ out of the solutions R_A, R_B based on the weight function z, specifying what area of which solution is being reused in the resulting solution. Note that we did not compare the performance of the two original solutions in the two regions. We rather have arbitrarily assigned one of the partitions of G_{sad} to one original solution. Therefore, we also construct the corresponding opposite crossing $\hat{R}' := (\hat{p}', \hat{H}')$ by using $\hat{z}$ instead of z as the weight function, where $\hat{z}(v) := 1 - z(v)$.

Note that we treat both representations R' and $\hat{R}'$ equally when they are then being optimized. This is because we do not see a good way to measure which original representation is advantageous for which of the two regions. This is why each crossing results in two new solutions, as mentioned above.

Tuning Parameters

For the remaining open questions, we decided to introduce the following three tuning parameters: the ratio of the running time which is used for crossings, the number of solution pairs which are crossed, and the initial temperature of a thread optimizing a crossing. For the latter parameter, we decided to express this as a multiple of the average temperature of the two original solutions which were used for constructing this crossing.

In a corresponding experiment in Chapter 6, we try different values for these parameters.

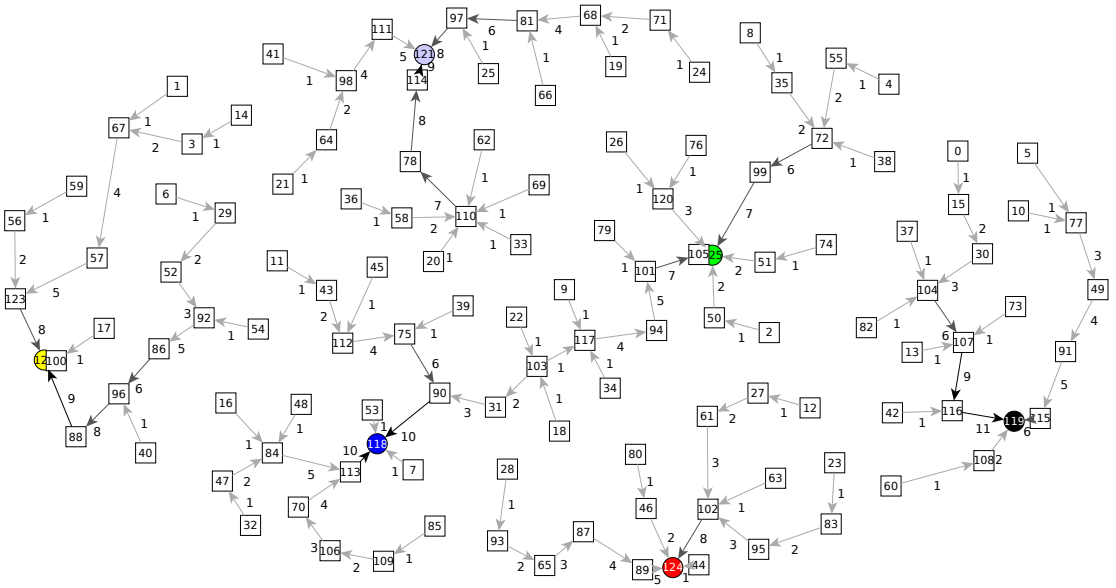


Figure 5.3: A representation and its corresponding solution. Substations are drawn as colored circles (the same colors are also used in the following figures). Node labels denote the potential values. Edge cuts are omitted to not complicate the illustration too much (they are not important for the issue we are discussing). Edge labels denote the cable throughput. Their shade indicates the cable type.

5.4 Improvement using a Two-Level Approach

As explained in Section 4.2, the Wind Farm Cable Layout Problem can be split into the Substation Assignment Problem and the Substation Cable Layout Problem, of which one instance per substation is to be solved.

In general, the idea of the following approach is to use the previously explained algorithm for the Wind Farm Cable Layout Problem, and additionally fine-tune the solution at the end of a Simulated Annealing computation. For this, we determine the substation assignment q of the solution, and split the problem into sub-problems for the fine-tuning. The fine-tuning is then done using our Simulated Annealing algorithm for each single substation. After fine-tuning, the cable layouts of the sub-problems can simply be merged to a solution for the full farm.

As already addressed in Section 4.2, splitting the set of turbines into one subset per assigned substation does not necessarily result in connected subgraphs induced by the node subsets. This is because the path of a turbine might use another turbine as an intermediate node, but the power of that turbine is not necessarily also routed to the same substation. While such a scenario is not really typical for real world wind farms, it can occur in our theoretical model, and therefore also in a result of our algorithm. However, in order to fine-tune the subgraph of a single turbine with an instance of the Simulated Annealing algorithm, it has to be connected. If a substation assignment induces only connected subgraphs, we call it *valid* for our fine-tuning. Otherwise, we call it *invalid*, but we try to fix it, i.e. find a similar assignment which is valid.

Invalid Substation Assignments

How the substation assignment of a solution looks like is not as straight-forward as it might first sound. This is because we "cancel out" flows in opposite directions when decoding the

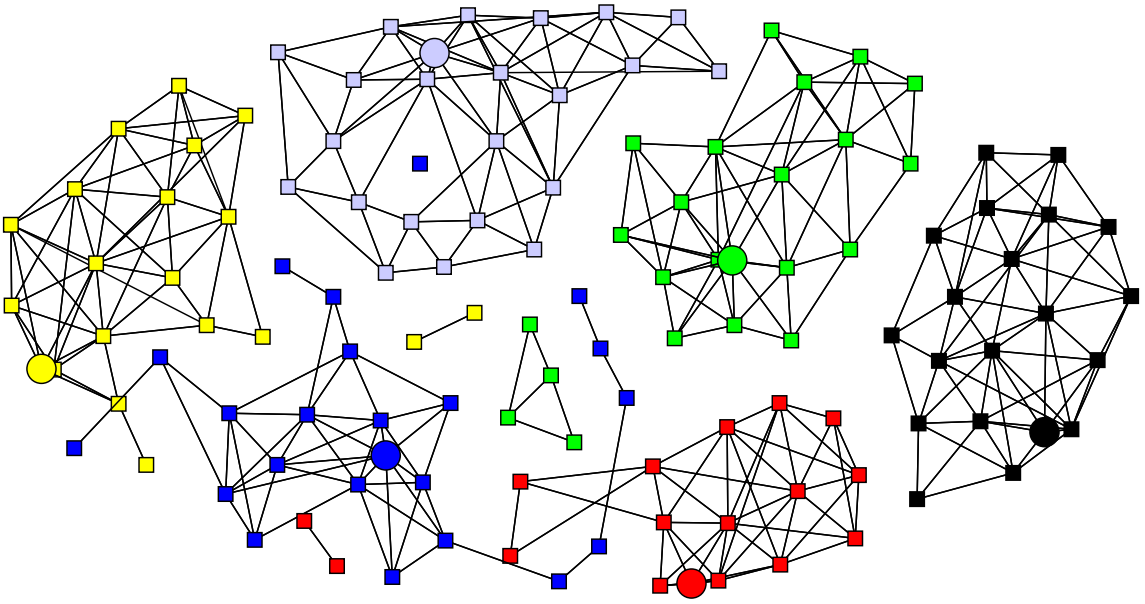


Figure 5.4: The substation assignment resulting from the representation from Figure 5.3. Each color corresponds to a substation. The edges are induced by the subset of the nodes. Note that the substation subgraphs are not connected, making the assignment invalid.

turbine path representation into the cable throughput representation (see our definition of the evaluation function in Section 5.1). We observed in our experiments, that very often the paths of two turbines use the same edge in the opposite direction. Intuitively, one might think that this does not happen, as both turbines use the same potential field to find the best next node during the path construction. However, as we do not allow the path to form cycles, in particular the predecessor node is not considered when determining the best next node.

To illustrate the resulting problem, in Figure 5.3 you can see the potential field of a representation as node labels. The resulting cables are drawn as differently shaded edges with their throughput (the absolute flow value) as labels. The colors in Figure 5.4 illustrate the substation each node was assigned (i.e. at which substation its path terminates). Note for example how the two disconnected nodes near the center, which are assigned to the yellow substation, are separated by some turbines assigned to the blue substation. At the same time, two nodes near the yellow substation are assigned to the blue substation. However, the resulting solution as visualized by the edges in Figure 5.3 contradicts to this situation (note that the two separated yellow nodes are connected to the blue substation).

Fixing Invalid Substation Assignments

As one possible attempt to fix this problem, we tried the following: for every node v in any subgraph G_v, which is disconnected from its corresponding substation, we find another node as a candidate for swapping their assigned substations. For this, we suggest to select the node $w \in N(v)$ (a neighbor in the original graph) which itself has the maximum number of neighbors already being in the subgraph G_v (i.e. maximizing $|N(w) \cap G_v|$). To apply the swap, we remove v from the subset of nodes of G_v and add it to G_w. At the same time we remove w from G_w and add it to G_v. The corresponding new sets of edges are the one induced by the subsets of nodes. In other words, we just pretend that v and w have been assigned to the opposite substations from the beginning.

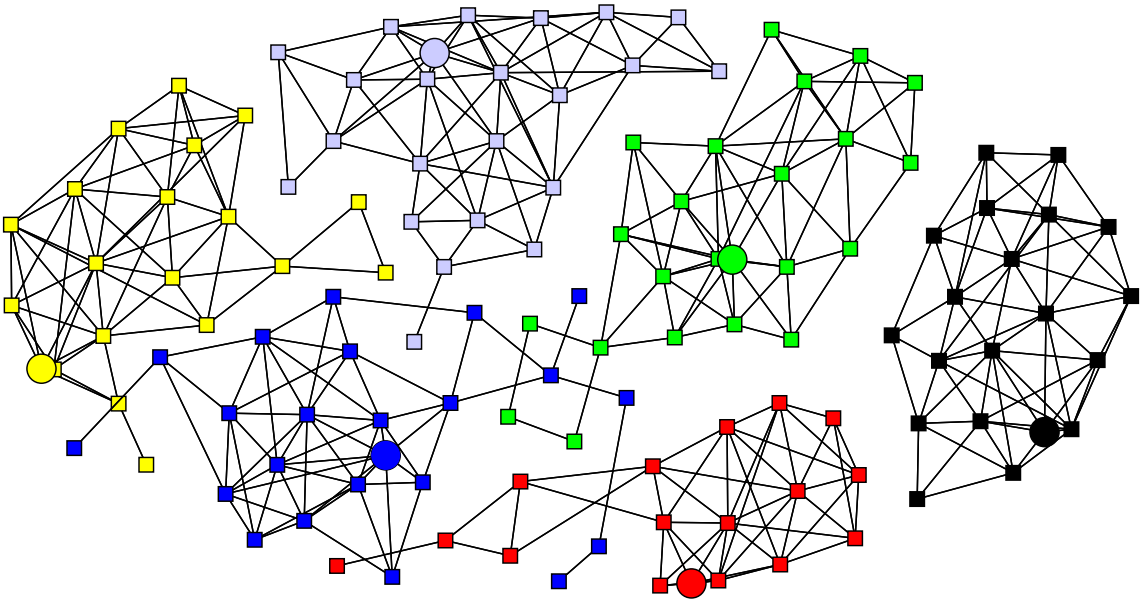


Figure 5.5: The valid substation assignment after fixing the invalid substation assignment from Figure 5.4.

This procedure is repeated for every disconnected node in any subgraph resulting from the substation assignment. The final result does not necessarily only contain connected subgraphs, as every swap not only modifies the connection of the node we want to fix, but at the same time disrupt existing connections in another subgraph. Figure 5.5 shows the result after four iterations of this procedure. We observe that the result is very different from the original assignment. Although this assignment is now valid, it is not suited for our two-level approach: experiments showed that when optimizing the individual substation cabling problems, their total cost does not improve the original results, and instead are much worse than that.

Yet we decided to leave this issue at that, and continue describing how the algorithm proceeds once a valid substation assignment was found. In the conclusion in Chapter 7, we propose a different method for splitting the wind farm cabling problem into individual substation problems.

Integration in Overall Algorithm

Similar to the evolutionary algorithm, we integrate our two-level approach in our existing algorithm as a separate phase. The majority of the total available computation time is used for running our existing algorithm, which finds relevant substation assignments. During this phase, for all solutions which caused an update of the best solution seen so far, the assignment and the cost of the best solution are recorded.

After the first phase, this list of substation assignment candidates is ordered by the cost of the best solution which has been seen by the Simulated Annealing algorithm. The best of them are considered for the second phase, in which they are fine-tuned. How many substation assignments are considered is again a configuration parameter. The substation assignments are split into their individual sub-problems, each consisting of a substation and the corresponding turbines.

Note that among the different substation assignments, the same sub-problem may occur multiple times. In these cases, the assignments are identical in that region, but differ elsewhere. Taking that into account, before solving the sub-problems, we identify equal

ones by their subset of nodes. Technically, this can be done using a bit vector, which can easily be hashed to allow fast lookups.

Future Work

In general, we find the two-level approach a good idea to solve the Wind Farm Cable Layout Problem, since each substation network can be optimized very well with our Simulated Annealing-based heuristic. However, the difficult part seems to find a good substation assignment, that is a partitioning of the original problem into smaller sub-problems.

For this, our idea was to find multiple such substation assignments in the first level, which seem promising to analyze furhter in the second level. We need to consider multiple assignments, since finding the best substation assignment in the first level would not reduce the original problem (as the cabling for each individual substation network needs to be considered just as well). However, this first level heuristic could be replaced with a whole different partitioning method, as it is independent of which algorithm is used for the second level.

Future work on this topic could focus on finding a suitable partitioning method for these substation assignments. When still using our Simulated Annealing approach for this, we believe that a variant of *Fuzzy C-means* clustering algorithm can be used to construct a substation assignment from the representation. For this, the node distance metric used in Fuzzy C-means could use the cables induced by the representation. A different method could be engineered based on multiple deep-first searches in the cable network, started at each substation and hitting each other at the partitioning borders.

Alternatively, we suggest to still use Simulated Annealing for the first level, but with a different representation: the substation assignment itself, making the above construction needless. Evaluating such a representation could then be done by solving the single substation problems it induces. An initial solution could be created with a generic partitioning or clustering algorithm, and the mutation could simply swap the assigned substations of two turbines. Basically, this method is a hierarchy of two different and independent Simulated Annealing heuristics.

6. Experiments

We implemented the heuristic from Chapter 5 which we now evaluate with experiments. In this chapter, we try to answer some interesting questions. First of all, we would like to know how well and how fast our implementation of the heuristic optimizes the Wind Farm Cable Layout Problem. Primarily, a comparison against solving the MILP model with a generic optimizer serves as a benchmark. Furthermore, we analyze how the properties of the input data influence the performance of our heuristic as well as the MILP. In particular, the input size is expected to be a major factor, but we also would like to find out how the performance correlates with other properties. For this, we picked two properties we found interesting to analyze in more detail.

One such property is the number of substations s: How much does an increasing number of substations influence the running time required to gain a solution of comparable quality? We expect that assigning turbines to substations plays a major role in the difficulty of our heuristic, and as such expect a notable correlation between the number of substations and the performance. We have to clarify which instances we compare with each other, i.e. how exactly we set their properties. When raising the number of substations s, how do we change the number of turbines t in order to create an instance which can be compared with the first? We decided to compare instances with a similar turbines-to-substations ratio t/s. As a result, it is natural that a larger number of substations results in a longer running time; however, if the substation assignment would not play a huge role, this running time would only increase linear in s. This is intuitive when thinking about simply splitting the problem into s sub-problems with a single substation each, which is possible when we know (or fix) the substation assignment.

Another interesting property is the *substation capacity tightness*. Our problem model allows instances in which the sum of all substation capacities is greater than or equal to the number of turbines. When being greater, we allow some flexibility when assigning the turbines to the substations. Similar to the previous expectations, we assume that a higher flexibility (i.e. when the substation capacities are *loose*) reduces the difficulty of finding a feasible and cost-optimal substation assignment.

We are also interested in whether or not our heuristic always ends in the same local optima when being run repeatedly with a different randomness. To analyze this, our implementation uses a pseudo-random number generator which is initialized with a seed, and thus shows a deterministic behavior depending only on the value of that seed. When running with different seeds, we see how the randomness influences the results. We compare

these results by their substation assignment, as this is the major property of different results.

We finally want to know whether our suggested improvements in Section 5.2 increase the quality of the solutions found by the heuristic.

Hypotheses

For all of these investigations we formulate some hypotheses which we then try to prove or disprove in the rest of this chapter using some experiments.

Before we evaluate the results and the behavior of our heuristic, we start with the following hypothesis which analyzes an interesting property of optimal solutions to the Wind Farm Cable Layout Problem:

(H1) In a wind farm in which turbines are placed quite uniformly over its area, longer transport cables interconnecting two turbines are rarely used in a cost-optimal cable layout. Similarly, when considering the restricted problem in which from each turbine a cable can only be laid to one of the nearest neighbors, the solution's cost is only larger by some negligible amount.

We continue with measuring the *performance* of our basic Simulated Annealing heuristic from Section 5.1:

(H2) Our basic algorithm finds near-optimal cable layouts for most realistic wind farms within half an hour on a regular desktop computer, and is thus applicable in practice.

(H3) The larger the number of turbines of the wind farm, the more time the computation needs to find results with the same relative cost difference towards the optimal solution. (Note: the truth of this hypothesis is quite obvious, but we are furthermore interested in *how much* more time is needed for larger instances.)

(H4) Similarly, the tighter the substation capacities are in the wind farm, the more computation time the heuristic needs.

The following hypotheses are used to analyze the *behavior* of our basic Simulated Annealing heuristic:

(H5) The results of our basic algorithm highly depend on the random seed, even for long running times, when identifying results by their substation assignment.

(H6) Problem instances with a single substation are easier to solve compared to instances with multiple substations, measured by the quality of the result over time.

(H7) In the basic algorithm, when the activity of a computation falls below a certain level, an improvement of the best solution seen so far is rarely found, i.e. the computation can be cut off at some point.

Finally, in order to discuss and evaluate the *improvements*, as introduced in Section 5.2, we formulate the following hypotheses:

(H8) Making the velocity of the temperature drop depend on the activity, as suggested by the formula $T' := (1 - \mu\tau)T$, has an advantage.

(H9) Recovering computations which did not improve their currently best solution for some large number of iterations — by resetting them to their best solution seen so far — improves the final results.

(H10) Computing the same problem using multiple threads often results in better solutions compared to a single thread, where the total running time is fixed[1]. The available running time is split among all threads evenly.

(H11) Near the end of the total running time, when replacing these computation threads with a crossing of the two most compatible and at the same time diverse solutions, even better results can be achieved.

Regarding the two-level approach, as discussed in Section 5.4, we did not find a good way to construct a substation assignment based on a representation of our Simulated Annealing algorithm. We therefore will not analyze the following hypothesis, however we still list it for reasons of completeness:

(H12) The two-level approach as suggested in Section 5.4 can be used to furhter improve the results.

Our Implementation

For the following experiments, we implemented the basic algorithm and the improvements we proposed in Chapter 5. For this, we chose the C++ programming language in the current standard version C++14 (ISO/IEC 14882:2014). Our implementation makes use of the C++ standard library primarily for containers and standard algorithms. The Qt Framework [qt] (version 5.5) is used for concurrency and for the GUI which we used for visual debugging. The Open Graph Drawing Framework (OGDF) [CGJ+] (version 2015.05) is used for input and output of graph files.

All experiments were run on a compute server which has four 12-core CPUs of the type AMD 6172, each clocked at 2100 MHz. The CPU has a 64-bit architecture, and the machine runs OpenSUSE 13.2 with Linux kernel version 3.16.7-35. The machine has 256 GiB memory installed.

We compiled our implementation with the GNU C++ compiler (g++, version 4.8.3) with maximum optimization level (`-O3`) and for the native architecture (`-march=native`). We linked against the release versions of the mentioned libraries. We linked the OGDF library statically and all other libraries dynamically.

Although our implementation has multi-threading support, when multiple computations are performed (as suggested in Section 5.2), each experiment runs on a single core. When computing multiple instances, they are scheduled in a very simple interlaced, round-based manner: in our implementation, each computation runs for a fixed number of iterations before it is paused, after which execution continues with the next computation thread.

The machine has 48 cores, so we execute multiple experiments on the same machine in parallel. However, in order to avoid that these executions influence each other, or are influenced by system processes, we never executed more than 45 experiments simultaneously.

Furthermore, we implemented the MILP model from Section 4.1 using the C++ binding of Gurobi Optimizer [gur]. Similar to our algorithm, the optimizer was run in single-thread mode for a fair comparison. As explained later, the MILP implementation serves as a benchmark for measuring the performance of our algorithm in the experiments.

We did not have time to implement the Substation Positioning Problem from Section 4.3. It is therefore not part of our experiments.

[1] Please note that when we talk about "multiple threads", the computation power is *not* increased. We do not talk about the number of physical threads available for computation, but rather about how many instances of the same computation, with a different randomness, are executed. The actual implementation may as well use only a single physical thread, which is used to execute the multiple computations of our algorithm concurrently, i.e. in an alternating manner.

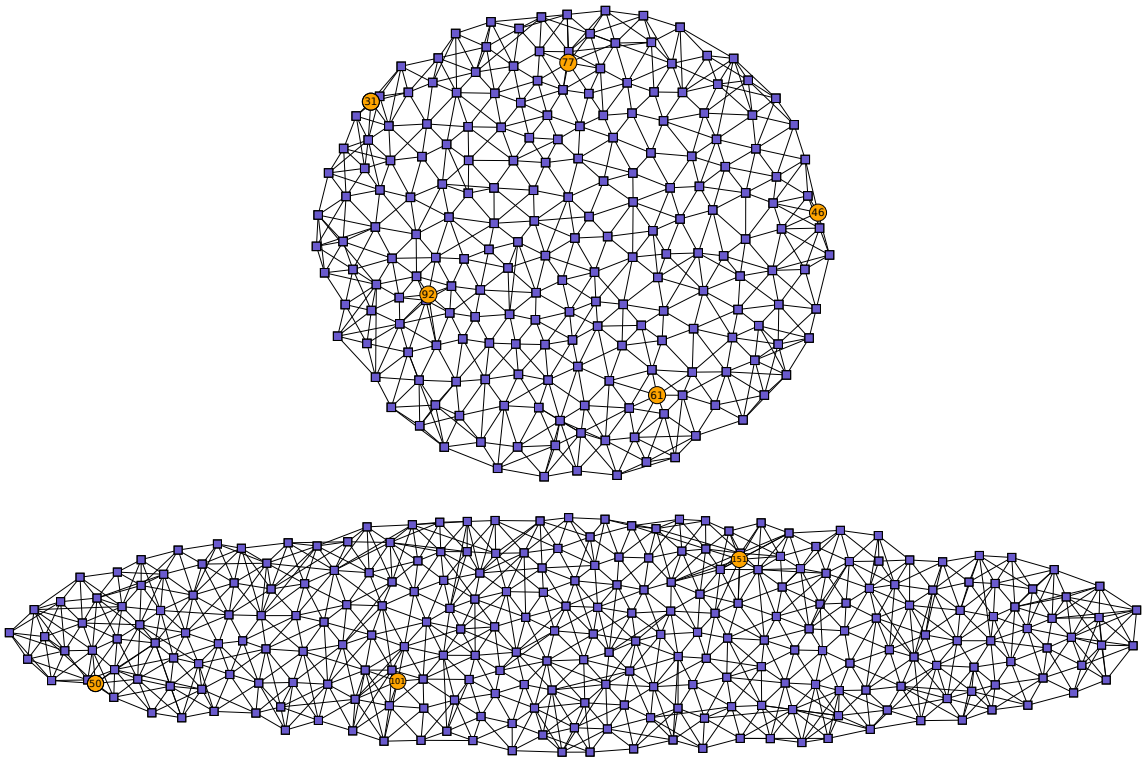


Figure 6.1: Two examples of generated instances with different values for β. The shape of the upper wind farm is almost a circle ($\beta \simeq 1$), while the shape of the lower one is a strongly biased ellipse ($\beta \simeq 0.2$). Turbine nodes are drawn as blue squares, and substations as yellow circles. Their label denotes the substation capacity.

6.1 Generating Random Instances

To automatically generate random instances for our thorough evaluation, we introduce numerical parameters which describe the most important characteristics of a typical wind farm. Based on these and a random seed, a deterministic algorithm computes the input data for our heuristics.

As already used throughout previous chapters, the parameters t and s describe the input size: we consider a farm with t turbines and s substations. We normalize the minimum distance between any pair of turbines to 1.

The shape of a farm is an ellipse. We introduce $\beta \in (0, 1]$ as the parameter describing the aspect ratio of the ellipse. When $\beta = 1$, the ellipse is a circle. The size of the shape is set such that its area equals t. Two examples with different shapes resulting from different values of β are shown in Figure 6.1.

Placing the turbines within this shape is done using Poisson Disc Sampling, a random point placement strategy in which all points have a minimum distance to each other (1 in our case). For this, random points are placed inside the shape iteratively. After generating a random point, the distance to the already existing points are checked against the minimum pairwise distance criterion. The point is then only accepted if all other points are far enough. As this process may take forever to complete (since towards the end, the scene is already full of points with no space left for additional points), after each rejection of a point the whole farm is scaled up by a factor very little larger than 1 (we found 1.00001 to work quite well).

The substations are placed in the same way with a minimum pairwise distance of $\sqrt{t/s}$.

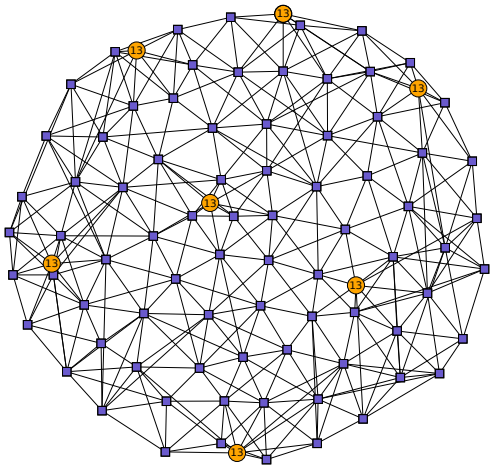


Figure 6.2: An example of a generated instance with $t = 79, s = 7$ and somewhat loose substation capacities ($\gamma \simeq 0.87$, resulting in $\sum_i m_i = 91$). Furthermore, as $\delta = 0$ in this example, every substation has capacity $m_{\text{avg}} = 13$. This combination of the parameters γ and δ results in some flexibility of the substation assignment.

The substation capacities are influenced by two additional parameters. The *substation capacity tightness factor* γ is the turbine-to-total-capacities ratio: when $\gamma = 1$, the sum of substation capacities equals the sum of produced power in the farm, i.e. the substations are fully saturated. We say, the capacities are *tight*. When being less than 1, less power is produced in the farm than the substations could support. This means that the assignment of turbines to the substations is more flexible than in the fully saturated case. The sum of substation capacities is thus given by t/γ.

The second parameter regarding substation capacities is δ, which describes the variance of the capacities among the substations: while the average capacity is $m_{\text{avg}} := t/(s \cdot \gamma)$, the allowed range of substation capacities is $[\,(1-\delta)\,m_{\text{avg}},\ \delta\,m_{\text{avg}}\,]$. The individual substation capacities are chosen randomly in this interval while making sure that their sum equals t/γ. To see the effect of this parameter, compare the instances in Figure 6.1, where $\delta \simeq 0.5$, with the instance in Figure 6.2, where $\delta = 0$.

The connections between turbines and substations, on which cables can be laid, are given as edges. Our instance generator can generate fully connected graphs, i.e. there is an edge for every pair of nodes (except from substations to substations). However, this often does not make much sense, as connections over longer distances are rarely done directly but instead turbines on the way are used as intermediate nodes. This is usually cheaper because the other turbines also need to be connected to substations, and the cables in use can transport power from multiple turbines. To put it short, optimal solutions tend to use shorter cables over longer ones.

For this, our generator proceeds in two steps. First, we add *first class edges* E_1 by connecting each node to its k nearest neighbors, measured with the euclidean distance. The parameter k is configurable, and we found $k = 6$ to work great. Note that a complete graph can be generated by setting $k := n - 1$.

Then, *second class edges* E_2 are added to make some shortcuts between any two nodes which are not directly connected but are fairly near to each other, such that laying a cable directly makes sense. For this, we introduce the parameter $\epsilon \in [1, \infty)$, which specifies the tolerance of the following approach. We found $\epsilon = 1.1$ to work great. We define $S \subset V \times V$

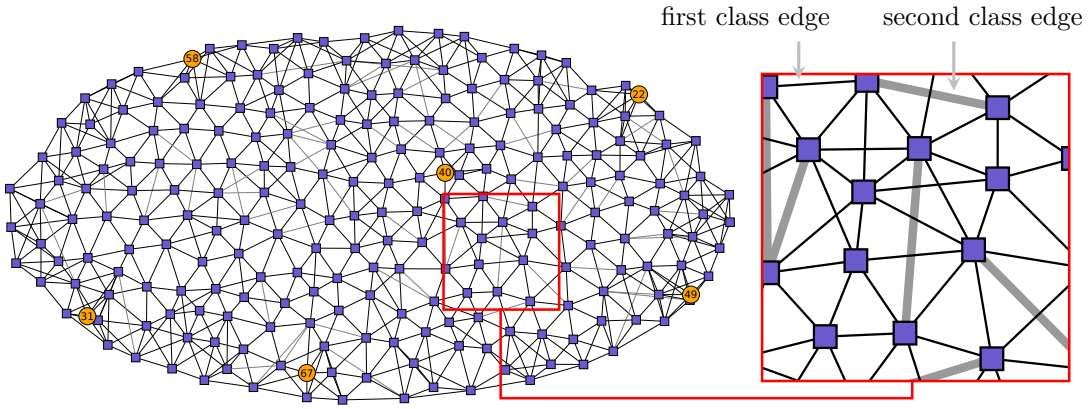


Figure 6.3: An example of a generated instance in which first class edges (each node is connected to its six nearest neighbors) are drawn black and second class edges (shortcuts with $\epsilon = 1.1$) gray. In the magnification, the shortcuts are drawn thick for better distinction.

Cable number	Capacity	Cost per length
1	5	20
2	8	25
3	12	27
4	15	41

Table 6.1: The four cable types we use in our experiments. The cost for a cable on an individual edge are computed as the product of the cost per length and the euclidean distance of the two nodes.

as the set of node pairs (v, w), which are not connected by a first class edge, but which are connected indirectly by a path of two first class edges, that is

$$S := \{(v, w) \in V \times V \ : \ (v, w) \notin E_1 \ \wedge \ \exists x \in V \text{ such that } (v, x) \in E_1 \wedge (x, w) \in E_1\}$$

Then, for any $(v, w) \in S$ we find the node $x' \in V$ resulting in the shortest such indirect connection, that is

$$x' := \arg\min_{x \in V}(d_{vx} + d_{xw})$$

If the length of the indirect way, when compared with a direct connection from v to w, is greater than factor ϵ (that is, if $d_{vx'} + d_{x'w} > \epsilon \cdot d_{vw}$), we add the shortcut (v, w) to E_2. An example showing these two different classes of edges is shown in Figure 6.3.

Although our algorithm can handle distinct sets of cable types for each edge, in practice we are given a fixed set of cable types which can be used on any edge. The instances we generate use the same set of cables for all edges. Also, we use the same set of cable types for all instances. Throughout our experiments, we use the cables from [Ber], which obtained their data from domain experts. The cable's costs per length and rating limits are shown in Table 6.1.

Sets of Instances

For our experiments, we generated multiple sets of instances. Our generator takes an interval for most parameters, from which it randomly chooses the actual parameter value. For example, a set of relatively small instances can be generated by telling the generator to

Set name	β min	β max	t min	t max	s min	s max	t/s min	t/s max	γ min	γ max	δ	k	ϵ	no. of instances
(T1) small/single	0.7	1	10	80	1	1	–	–	–	–	–	6	1.1	500
(T2) small	0.7	1	10	80	2	7	10	20	0.83	1	0	6	1.1	500
(T3) medium	0.5	1	80	200	4	10	10	20	0.83	1	0	6	1.1	1000
(T4) large	0.4	1	200	1000	10	40	10	50	0.83	1	0	6	1.1	1000
(T5) medium/complete	0.5	1	80	200	4	10	10	20	0.83	1	0	∞	–	1000

Table 6.2: The sets of instances we use for our experiments. (T1)–(T4) are our basic instance sets intended to resemble relatively realistic wind farms and used in most of our experiments. The advanced set of instances (T5) is needed for a special experiment.

choose $t \in [10, 80]$. Furthermore, the generator is told the number of instances to generate, and for each instance a new parameter set is chosen randomly from the given intervals. The parameter values are written into a table which makes it possible to later analyze subsets of the instances by sub-dividing the intervals more granularly.

We generated the following sets of instances (see Table 6.2). Primarily, we needed different sizes of inputs (T2)–(T4). We also need some instances with only a single substation to analyze if these problems are easier to solve — for this we focused on small instances (T1). For these basic sets of instances, we choose relatively realistic values for most parameters. Furthermore, we want to analyze the behavior of our algorithm as well as the linear program when the set of edges are not restricted in advance. For this, we created a set of instances with complete graphs (T5). The turbines and substations in these instances are the very same as in set (T3). The only difference of the instances is that the edges, i.e. the pairs of nodes which can be connected directly with a cable, are not restricted at all.

6.2 General Observations

Before analyzing the performance and the behavior of our heuristics, we need *reference solutions* against which we can compare any results of our algorithms. For this, we chose to use the results of the linear program we proposed in Section 4.1.

Results of the Linear Program

To compute the reference solutions, we used the *Gurobi Optimizer* [gur], which we run for one hour for each instance. Note that we executed the program in single-threaded mode on the same machine on which we also run our own algorithm for a fair comparison.

Please note the following important fact about these reference solutions. They cannot be treated as the optimal solution, as the optimization process has been interrupted after the time limit of one hour. Also, guessing the cost of the optimal solution from the reference solution, such as by some fixed error factor, is impossible. To reason this, we need to understand how we find that reference solution: the optimization software regularly improves both an upper bound solution as well as a lower bound relaxation. The costs of these two have a quite large gap to each other, which is decreased during the optimization process. Our reference solution is what Gurobi outputs as the upper bound after one hour, since that is the best currently known valid solution to the integer linear program. We observe that when continuing the computation further, the lower bound increases over time, reducing the gap, while the upper bound decreases much slower. For an exemplary visualization of the lower and upper bound over time see Figure 6.4.

Applied to our scenario, this can be interpreted as follows: the optimizer finds a feasible solution, which is near-optimal, within a practical running time of about one hour, even

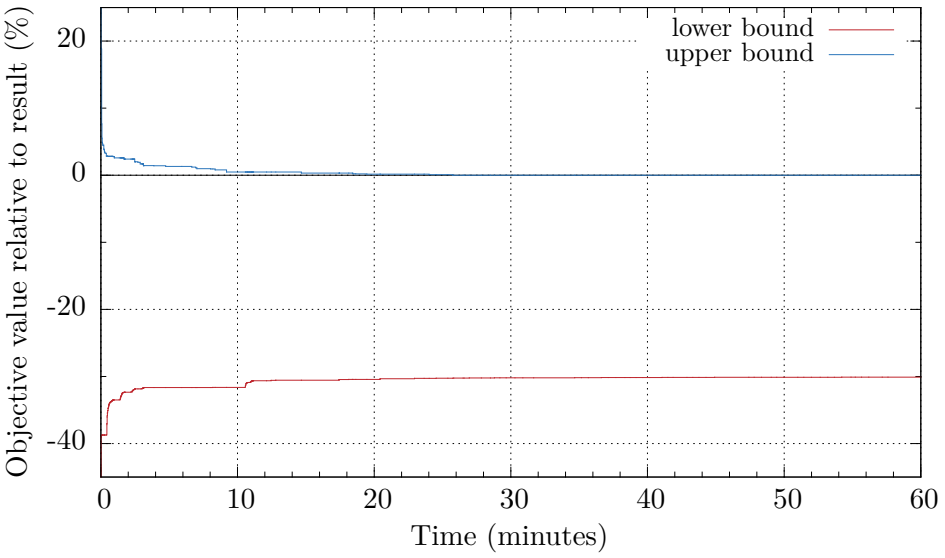


Figure 6.4: Visualization of the upper and lower bound of Gurobi Optimizer over time at the example of one instance from (T3) with 140 turbines. The computation is stopped after one hour, which we use in our experiments as the reference solution. The vertical axis expresses the cost of the solution at each point in time relative to this reference. Positive percentages mean higher cost, while a negative percentage means lower cost compared to the reference. For a vertically magnified visualization of the upper bound, see Figure 6.11.

for large wind farms. To reduce the difference to the absolute optimal solution, or to prove that this is already reached, it takes a very long time. This total time seems to be exponential in the size of the input (as indicated by the NP-hardness of the problem), but is impractical to measure.

The problem here is that it is difficult to create fair reference solutions (or even only a guess for its cost): for larger instances, the relative difference of the upper bound towards the unknown optimal solution seems to be larger than that of smaller instances, when running the optimizer for a fixed running time. We can not know how this difference depends on the size (or any other property) of the input, as that would require to finish all computations which takes an impractical amount of time (under the assumption that the time rises exponentially with the input size).

We observed the following gaps between the upper and the lower bound as found by the Gurobi Optimizer, depending on the instance sizes: for the very small instances from set (T1) with up to 20 turbines, the gap was below 20 % in most cases (see Figure 6.5). For all instances from the sets (T1, T2), the average gap was about 22 %. For the medium-sized instances (T3), it averages to 30 to 31 %, and for the large instances (T4), it reaches 32 % (see Figure 6.6). We observed that for most instances with only up to 13 turbines, the optimizer reaches gaps below 10^{-4}, at which the computation is cut-off before reaching the one hour time limit[2] (see Figure 6.5). We see that the time Gurobi requires to reach that gap limit depends on the input size: for only 10 turbines, the solution is found very quickly. The time is about quadrupled when going from 10 to 11 turbines, and again about quadrupled from 11 to 12 turbines.[3]

[2]This is the default behavior of Gurobi Optimizer, which can be overridden. However, such a low gap means that the absolute optimal solution can not cost less than 99.99 % of the solution found.

[3]Although this indicates an exponentially growing running time, we do not have enough data available to analyze this further, as the time required to solve larger instances would be very long.

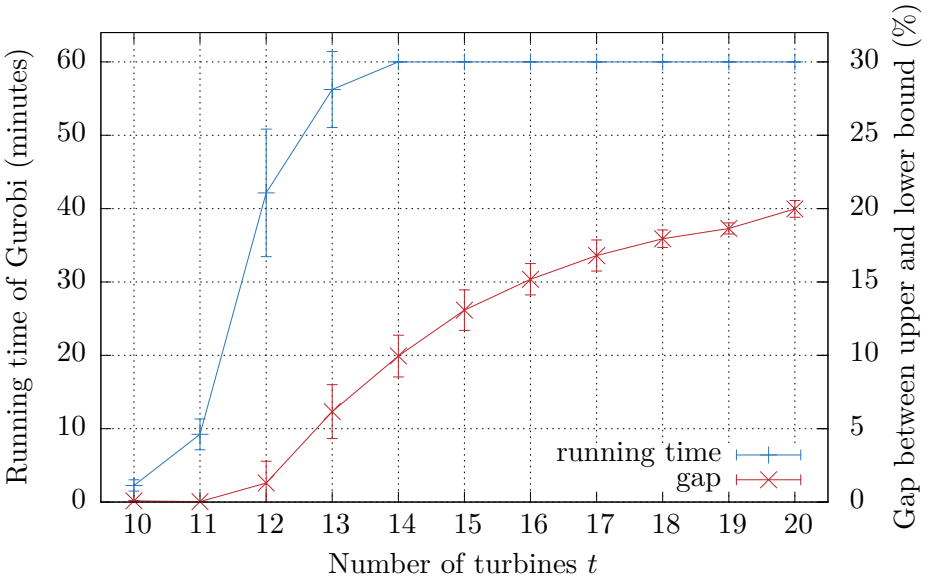


Figure 6.5: Visualization of the running time of Gurobi Optimizer and the gap between the upper and lower bound. The running time is limited to one hour. Here, we only consider the very small instances from (T1) with up to 20 turbines. Gurobi finishes before the time limit for most instances with 13 turbines or less.

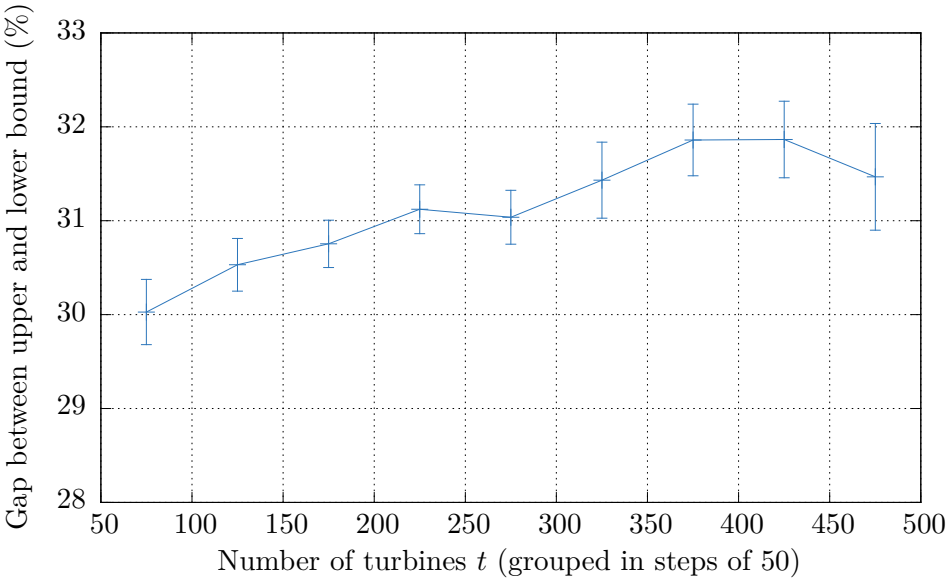


Figure 6.6: Visualization of the gap when solving the instances from sets (T3) and (T4) with Gurobi Optimizer with a time limit of one hour. The instances are grouped by their size in steps of 50.

Results for Small Instances

With the use of the following experiments, we analyze the performance of our basic Simulated Annealing heuristic as explained in Section 5.1.

We executed our implementation on the small instances (T1, T2). For this, we set the time limit to 2 minutes, while choosing 30 minutes for the medium-sized and large instances (T3, T4). The temperature is initialized with $T_0 := 0.01$. For the shorter running times,

we set $\tau := 10^{-5}$, while for the longer running times, we set $\tau := 10^{-6}$. The idea is that the temperature curve can be lowered more slowly if more time is available.

For all following experiments, we chose the dynamic temperature curve, because it seems to be a good improvement, as we will see later in detail. We also enable the "recover" mechanism which winds back the computation in case it was not improved for more than a configurable number of iterations. We set this parameter to 10^4. All other improvements such as multiple instances and the crossings are disabled.

We observe that for the instances with a single substation (T1), our algorithm outperforms the reference solution after two minutes by some small percentage for 48.2 % of all instances, that is about every second one. In these cases, the relative difference of the solution's cost to that of the reference solution averages to -0.44 % (we use negative percentages to indicate a result better than the reference solution). In the other cases, our algorithm performs 1.77 % worse than the reference in average.

Very similar results are observed for the sets of instances with multiple substations, but with the same average size (T2): again, our algorithm outperforms the reference in roughly the half of all instances (51.4 %). In these cases, the average difference of the cost is -0.29 %, and in the cases in which our algorithm performs worse it is 0.41 %.

Note that until here, we did not tune the parameter τ. This is somewhat critical for the performance of the algorithm, as we will see below.

Results for Medium-Sized and Large Instances

For larger instances such as the ones from sets (T3) and (T4) are more difficult to solve by our algorithm. When using the same value for τ as before, that is $\tau = 10^{-6}$, in only 7.9 % of all instances our algorithm outperforms the reference solution. The problem here is that for larger instances, each iteration takes more time (especially the evaluation function). This means that less iterations can be performed in the same amount of running time. For example, within half an hour on our machine up to 35 million iterations can be performed for an instance of size $n = 50$. For $n = 100$, within the same time we can only compute 17 million iterations, and for $n = 200$ that are only 8 million. The set of instances (T4) contains wind farms with up to 500 turbines, for which only 3 million iterations are performed within 30 minutes on our machine.

In turn, the value of τ needs to be increased for larger instances such that the temperature curve reaches a point which is cool enough for the Simulated Annealing procedure to be effective. However, if τ is set too high, we risk that the temperature curve falls too quickly such that the algorithm does not have enough time to explore the solution space adequately.

The value for parameter τ has to be tuned very carefully, as it is crucial for a good performance of the algorithm. We performed experiments to find the optimal value for this parameter. We found that a value of at least $\tau = 10^{-5}$ should be chosen for the instances in set (T3), and even higher values should be preferred for the large instances from (T4).

More precisely, we performed the following experiment to find the optimal value of τ depending on the instance size. For this experiment, we chose the sets of instances (T3) and (T4). For τ, we tried the values $2^k \cdot 10^{-5}$ with $k \in \{0, \ldots, 4\}$, that are five different values of which each is twice as large as the one before. Again, we measure the performance of a solution by the relative difference of the cost towards the reference solution of Gurobi.

We grouped the instances by their size in steps of 50. For each group and each value of τ, we aggregated the average performance and its standard deviation. The curve of each value of τ is visualized in Figure 6.7.

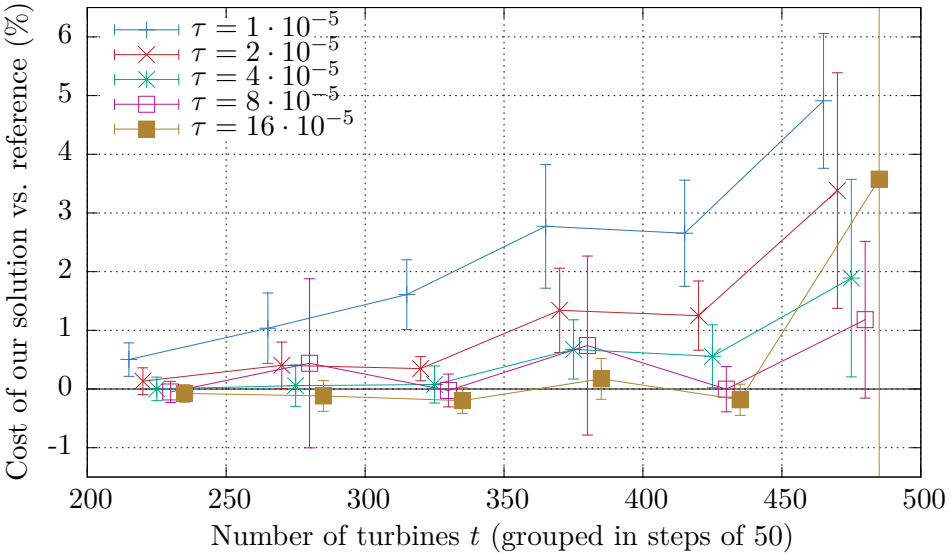


Figure 6.7: Plot showing the performance of our algorithm for different input sizes and different values of τ. The standard deviation is illustrated using the vertical bars, and they indicate that instances with more than 450 turbines are difficult to solve with our algorithm for any value of τ within the given time frame of 30 minutes. The aberration in the data point for this size and $\tau = 16 \cdot 10^{-5}$ comes from one instance which freezes too early. It has a cost 74 % larger than that of the reference solution. In most other cases, $\tau = 8 \cdot 10^{-5}$ or $\tau = 16 \cdot 10^{-5}$ works pretty well.

Results for Complete Graphs

Recall that our generated instances, as stated above, had been created with a restricted set of edges, i.e. not all pairs of turbines can be connected with a cable directly. We solved these instances with both Gurobi and our heuristic, and compared their results against each other. However, this is already a restriction of the solution space we made when generating the instances. The set of edges is a lot larger when we allow cables on all pairs of nodes, i.e. when considering a *complete graph* with $E = V \times V$ as the input.

However, the way we restricted our edges in the generator makes sense when looking at real wind farms, in which longer cables are not used whenever an indirect connection has only a detour of negligible length, and even for longer detours. The reason is usually that the turbines which lay in between such a longer distance also need to be connected somehow, and sharing the same cables for transporting power of multiple turbines usually is cheaper than laying multiple cables on the same or similar paths.

We made the following experiment in order to analyze this. We executed both the Gurobi Optimizer for the MILP as well as our algorithm on the two sets of instances (T3) and (T5). These instances only differ in their set of edges: (T3) only contains edges connecting each node to its six nearest neighbors, plus some shortcuts. The instances in (T5) are the corresponding complete versions of the instances from (T3).

For most instances (82 %), the cost of the MILP solutions are worse for the complete graph, compared to the cost of the solution for the corresponding restricted graph. However, for some instances (18 %), the opposite is true. When assuming that Gurobi does not optimize the complete versions faster than the restricted ones[4], this indicates that restricting the

[4]The opposite assumption is more realistic, i.e. that Gurobi optimizes the instances with less edges faster than the complete graphs.

edges impair the quality of the solution. However, in the cases when the solution for the complete graph was cheaper after running Gurobi for one hour, the difference in the cost averages to only 0.02 %. In the extreme case among our thousand instances the solution for the complete graph is 0.97 % cheaper. We conclude that the impair of restricting the set of edges is therefore negligible, confirming hypothesis (H1).

On the other side, our algorithm has a stronger issue with solving complete graphs. Again, we ran the algorithm for the same amount of time (half an hour) on both types of instances and compare the cost of the final result. In average, for complete graphs the cost is 2.45 % higher than for the restricted graph. This means that more time would have been required in the case of the complete graph in order to gain a solution costing the same (or less). Therefore, we conclude that it definitely makes sense to restrict the edges in advance, as it improves the computation performance significantly while affecting the quality only negligibly.

Comparing Tight vs. Loose Substation Capacities

Recall that our generator allows creating instances varying in the parameters within a given interval. We created instances which vary in the tightness of the substation capacities from $\gamma = 0.83$, in which case the capacities add up to 20 % more than the number of turbines, to $\gamma = 1$, in which case the sum of the capacities equals the number of turbines, leaving no flexibility in their saturation. With hypothesis (H4) we claim that there is a correlation between the capacity tightness γ and the difficulty of solving the instances.

We measure the difficulty in two ways: first, we analyze the gap between the upper and lower bound as found by the Gurobi Optimizer after one hour. This indicates how difficult an instance is to be optimized by this software. For our algorithm, we use the relative difference between the result of our algorithm and the reference solution, that is the upper bound found by Gurobi Optimizer. Again, we run our algorithm for half an hour for each instance.

For these experiments, we use the instances from set (T3), i.e. the medium-sized wind farms with 80 to 200 turbines. Again, we run our algorithm for 30 minutes. We use the parameter value $\tau = 8 \cdot 10^{-5}$, as we found that this is the optimal value for most instances of that size.

We visualize the results using scatter plots (see Figure 6.8). In both plots, the points are placed horizontally by their looseness of the substation capacities (γ): on the left, the capacities are tight, and on the right they are loose. The upper diagram shows on the vertical axis the gap of solving the MILP with Gurobi Optimizer. In the lower diagram, the vertical axis corresponds to the relative difference of the cost after running our basic algorithm for 30 minutes to the reference solution: near the top, the instances are more difficult to solve by our algorithm, while near the bottom they are easy. For instances placed below zero our algorithm outperforms the linear program.

For both diagrams, we are interested in the correlation coefficients of all data points to draw conclusions about how the properties correlate to each other. In addition to the data points, we see a linear fitting curve visualizing the correlation between the two axes. The linear correlation coefficient of the upper diagram is -0.04, indicating no significant correlation between the capacity tightness and the gap of Gurobi's upper and lower bound. In the second diagram, the correlation coefficient is 0.18, indicating a weak linear correlation of the capacity tightness and how much worse our algorithm performs compared to the reference solution. Although this correlation coefficient is still quite low, the diagram furthermore tells us that the instances for which our algorithm performs worst tend to be the ones with tight substation capacities.

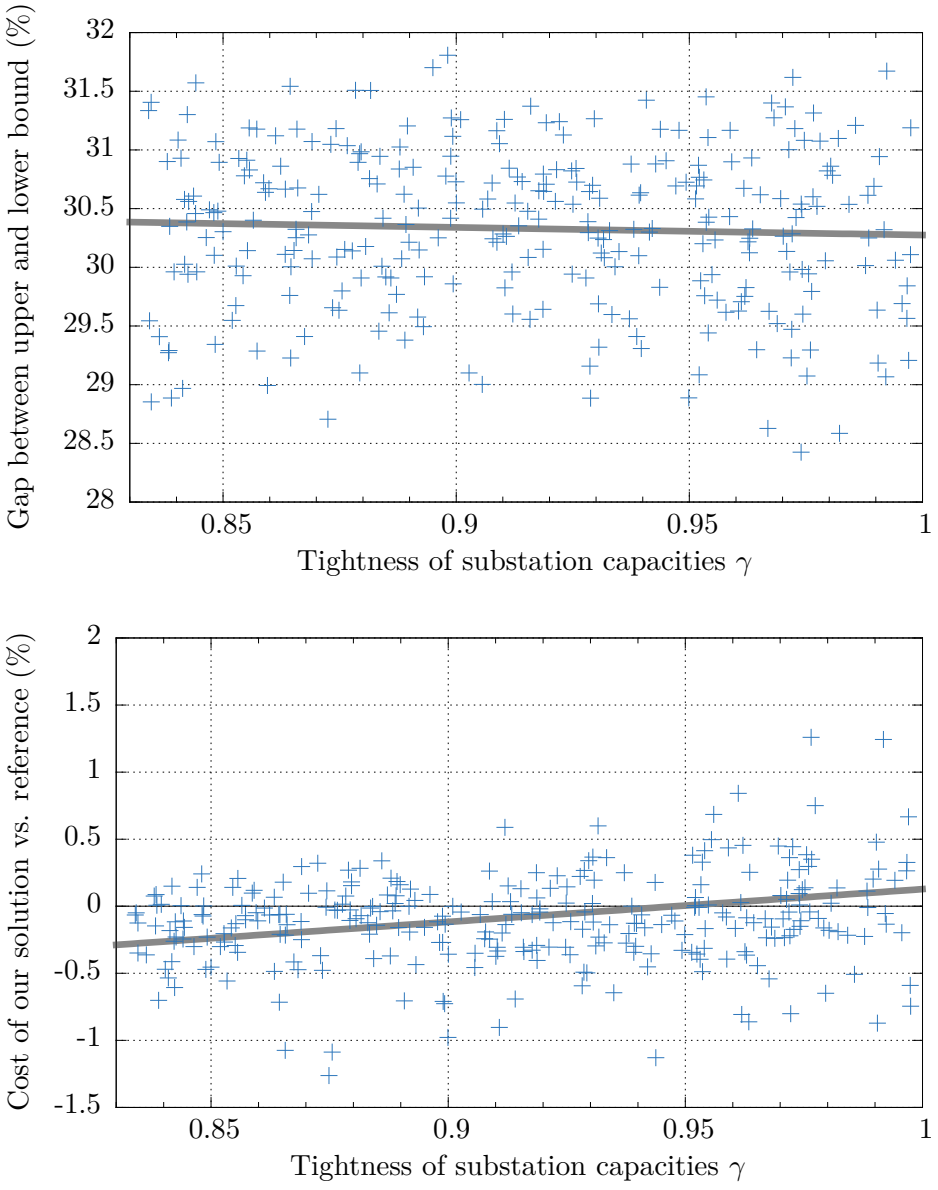


Figure 6.8: Scatter plots of the instances from set (T3) with the substation capacity tightness (γ) on the horizontal axis. The upper diagram shows the gap of the MILP. The lower diagram shows the relative quality of the solution found by our algorithm, where a positive percentage means our algorithm is worse, and a negative percentage means our algorithm is better than the reference solution.

We now discuss these results, which turns out to be rather difficult. As noted before, when interpreting the intermediate results of a Gurobi minimization, in general it is not possible to draw conclusions about the quality of the upper bound solution, i.e. its difference towards the actual optimal solution. We only know its difference towards some lower bound. However, when assuming a correlation between the quality and the gap, it seems that instances with tight capacities are not significantly more difficult to optimize by Gurobi than the ones with loose capacities.

Clearer conclusions can be drawn from the results visualized in the second diagram. Here, the correlation we observe indicates that our algorithm finds better results in the same time for instances in which the substation capacities are loose, compared to the tighter ones. In particular, for most instances in which $\gamma < 0.85$ our algorithm outperforms the

reference solution (as the majority of the data points are below zero), while for $\gamma > 0.95$ the average performance of our algorithm is worse than that of Gurobi. Also, cases in which our algorithm performs more than 0.5 % worse than Gurobi are only found for $\gamma > 0.9$. Together, this indicates the truth of hypothesis (H4).

6.3 Analyzing the Behavior of the Basic Algorithm

In the following, we analyze the hypotheses (H5)–(H7), concerning the behavior of our basic algorithm as explained in Section 5.1.

How the Random Seed Influences the Results

In hypothesis (H5), we claim that the random seed has a great influence on the final result of our Simulated Annealing algorithm, as proposed in Section 5.1.

To analyze this experimentally, we run the basic algorithm multiple times on the same instances, but with 50 different random seeds each. For each instance, we then count the number of distinct substation assignments of each solution, and divide this by 50. A value of 1 then means that every computation resulted in a different substation assignment, while lower numbers tell us that multiple computations show the same assignment in their result.

We used relatively small instances for this experiment, so we chose to use instances from set (T2). We furthermore decided to compare the results of two different total running times: 2 minutes and 30 minutes. Since the experiment takes a lot of time due to the multiple seed values, we only used the first 300 instances of (T2) for the shorter, and only 50 instances for the longer experiment. Furthermore, to adjust the temperature curve for the different running times, similar to the first experiments on page 43, we set $\tau = 10^{-5}$ in the case of 2 minutes, while in the other case we set $\tau = 10^{-6}$.

In Figure 6.9 we see that for a low number of substations, the number of different substation assignments is relatively low: the average value is 0.33 for 2 substations. For a raising number of substations, the number of different substation assignments tends to 1. We

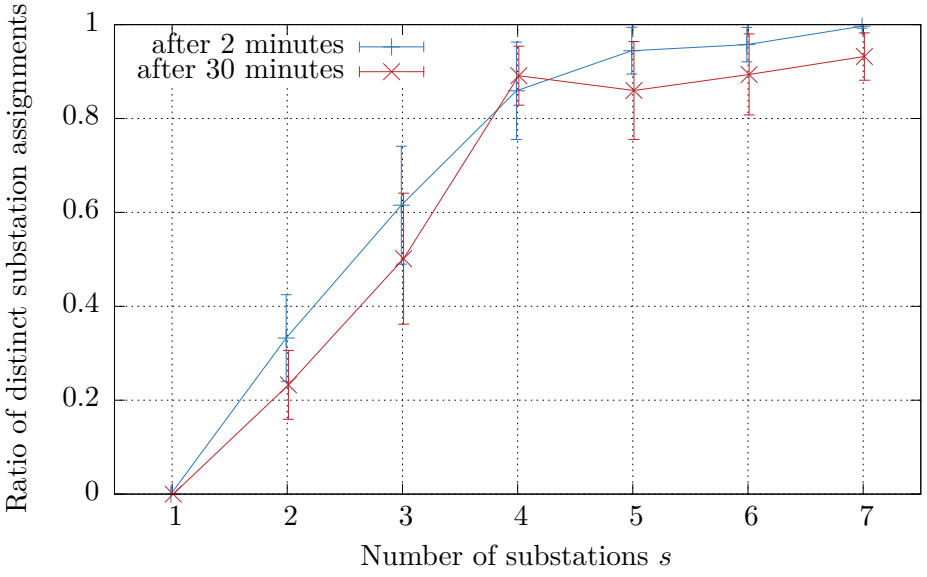


Figure 6.9: Plot showing how the ratio of distinct substation assignments per instance depends on the number of substations. Even for longer running times, the majority of computations end with different results.

Number of substations s	Interval for number of turbines t	Interval for ratio t/s
1	$[10, 15)$	$[10, 15)$
2	$[20, 30)$	$[10, 15)$
3	$[30, 45)$	$[10, 15)$
4	$[40, 60)$	$[10, 15)$
1	$[15, 20]$	$[15, 20]$
2	$[30, 40]$	$[15, 20]$
3	$[45, 60]$	$[15, 20]$
4	$[60, 80]$	$[15, 20]$

Table 6.3: The eight categories of instances we compare with each other for analyzing (H6).

furthermore observe that for the longer running times of 30 minutes, the effect is not as strong, but still very noticeable: for 7 substations, which is the maximum in this experiment, we have an average value of 0.93.

This is a strong indication that hypothesis (H5) is true.

Comparing Single vs. Multiple Substations

In our hypothesis (H6), we claim that problem instances with a single substations are easier to solve than those with multiple substations. For this, we compare instances with a similar turbines-to-substations ratio t/s. We chose to define two intervals for this ratio, which we analyze separately: instances with 10 to 15 turbines per substation, and the ones with 15 to 20 turbines per substation.

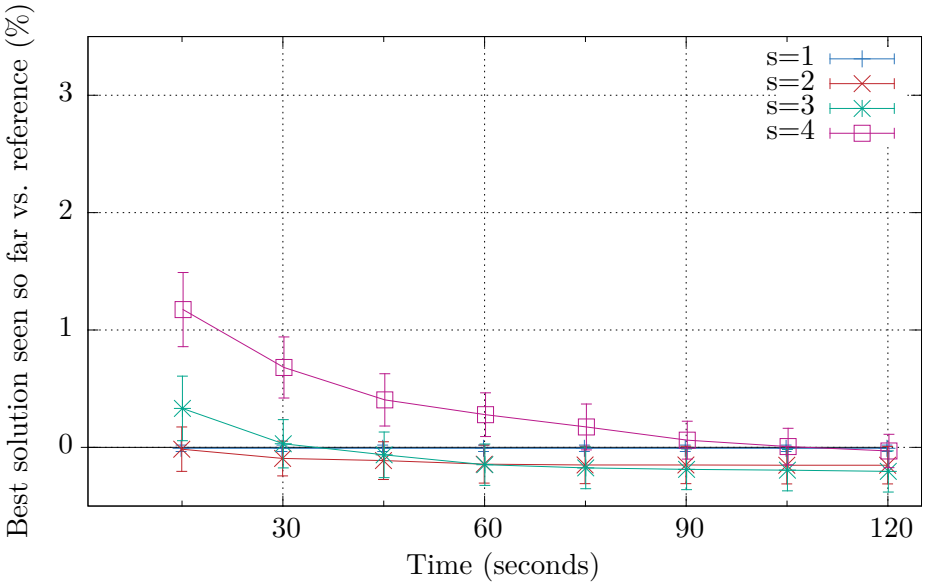
For this experiment, we chose the sets of instances (T1) for the single-substation case, and (T2) for the multiple-substations case. As both are limited to 80 turbines, and since we limit t/s to 20 as explained above, we only consider cases with up to four substations. Together with the above splitting, this results in eight categories of instances we compare to each other in this experiment (see Table 6.3).

The experiment to analyze the hypothesis is carried out as follows. For all instances, we run our algorithm for two minutes like in the previous experiment. We regularly check the current cost from each execution in intervals of 15 seconds. This is the cost of the best solution seen so far within the given time frame (i.e. this value never increases for a single execution). This leads to eight samples at different points in time for each execution. We compare these costs against the reference solution as a relative percentage, expressing how much more the current solution costs compared to the reference. Note that negative percentages can appear in this dataset, as our algorithm sometimes produces better results than the linear program.

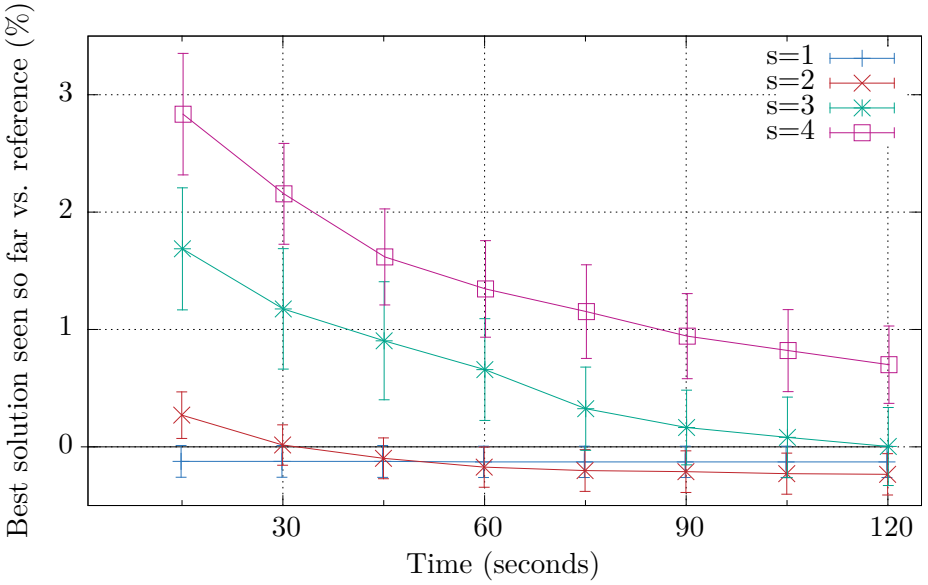
For each category and each of the eight points in time, we statistically aggregate these percentages of all instances within that category to an average value and a standard deviation. We finally plot this data with different curves over time (see Figure 6.10).

Interpreting these plots, it is quite obvious that for a larger number of substations, our algorithm does not reach the reference solution as quickly as for a single substation. The most straight-forward reason is the fact that the algorithm not only needs to find the exact path for each turbine to a substation resulting in the cheapest cabling, but also to *which* substation each turbine is to be connected in the best case.

This strongly indicates the truth of (H6). It is the primary motivation for our two-level approach we introduced in Section 5.4, in which we split the problem into the task of



(a) The results for the instances with $10 \leq \frac{t}{s} < 15$



(b) The results for the instances with $15 \leq \frac{t}{s} \leq 20$

Figure 6.10: Average performance of our algorithm over time for the eight different cases listed in Table 6.3.

finding a good substation assignment and solving multiple single-substation sub-problems, which are each relatively simple to solve.

The Activity as an Indicator for Dead Threads

In hypothesis (H7), we claim that once the activity of a thread falls below a certain threshold value, continuing the computation only negligibly improves the solution. This means, the thread can be considered dead and it can be stopped at that point, leaving more computation power for other non-dead threads.

We found it very hard to analyze this hypothesis. At the first sight, it seems to be intuitive that when the activity is very low, only little improvements can be found. This is because

a very low activity means that the probability of accepting a worse solution is very low (as it measures exactly that, but averaged over time). This makes a transition to a worse solution unlikely, and only leaves the possibility to accept better neighboring solutions. However, the neighborhood of a solution is limited by the way the mutation function is defined.

On the other hand, when a computation reaches this state, one reason is that the parameter τ was simply chosen too high, resulting in very low temperatures in the later stage. Although this is somehow the idea of Simulated Annealing, a too quickly falling temperature curves makes it hard to globally explore the search space in the early stage. The solution would be to use a larger value for τ. In this case, throughout the whole computation the probability that a better solution can be found in the future is significant. This means that the computation should not be stopped too early.

Therefore, we did not use the activity as an indicator that the computation can be cut off.

6.4 Evaluation of Our Suggested Improvements

In the following experiments, we analyze how our improvements suggested in Section 5.2. These include the dynamic temperature curve, the idea to perform multiple computations with different random seeds, as well as a strategy to recover from bad local optima. Finally, we suggested an approach similar to an evolutionary algorithm, in which crossings of good solutions are performed.

Dynamic Temperature Curve

One of our suggested improvements is to use a dynamic temperature curve in which the temperature change is proportional to the activity μ of the computation. The activity is the exponentially smoothed probability of accepting worse solutions, independent of the choice of accepting them. With the hypothesis (H8), we question if such a curve brings an advantage for the solution's quality. Figure 6.11 shows both the standard and the dynamic temperature curve, as well as the resulting performance curves over time for an instance from set (T3).

Before discussing the experiments to analyze this hypothesis, we compare the mathematical behavior of the two different temperature curves, and what this means for the choice of the parameter τ. In the original Simulated Annealing heuristic without our suggested improvement, the temperature delta which is subtracted after each iteration is

$$\Delta T_{\text{standard}} = \tau \cdot T$$

This leads to an exponentially decreasing temperature curve, resembling the physical process. We call this the *standard temperature curve*. In the case of the dynamic temperature curve we suggested, the temperature delta is

$$\Delta T_{\text{dynamic}} = \mu \cdot \tau \cdot T$$

When comparing those two formulas, in the case of the dynamic temperature curve, a larger value should be chosen for the parameter τ, as the factor μ which was added in the formula drastically decreases the resulting temperature delta. In early experiments, we found that τ should be approximately two orders of magnitude larger for the dynamic curve to result at about the same final temperature after half an hour. Of course, this is only an initial guess, and the parameter should be tuned appropriately.

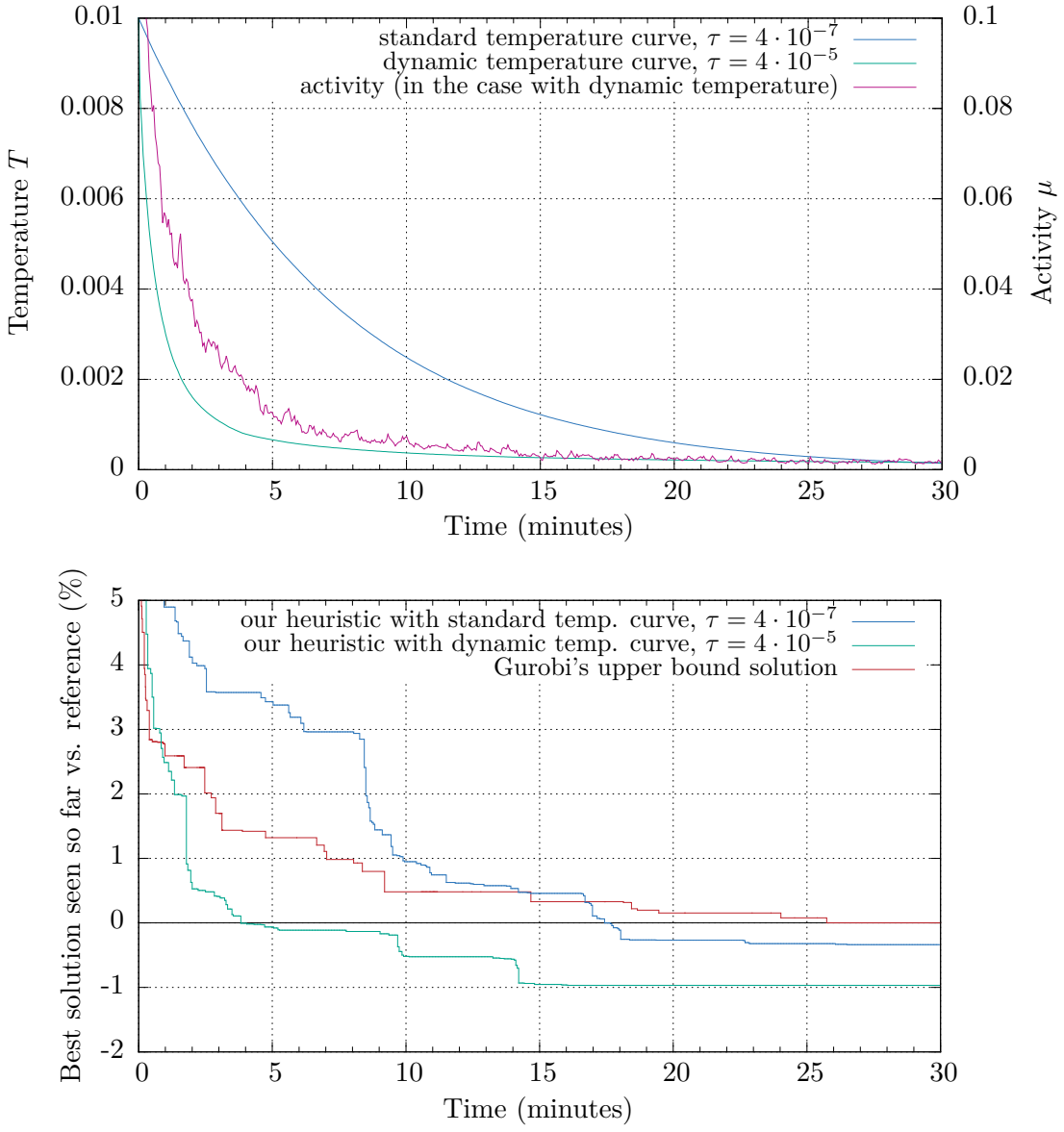


Figure 6.11: The standard and dynamic temperature curve at the example of an instance from set (T3) with 140 turbines and 8 substations. The upper diagram is the same as in Figure 5.2, which shows the temperature and activity over time. The lower diagram shows the cost over time for the two different experiments together with the upper bound solution of Gurobi Optimizer.

In the following experiment, we try to find the optimal value for τ for both the standard and the dynamic temperature curves. For this experiment, we chose to use the instances from sets (T3) and (T4) with up to 400 turbines. Again, we group these instances by their number of turbines in steps of 50. For both temperature curve types, we chose different values for τ. For the dynamic temperature curve, we try the five values $2^k \cdot 10^{-5}$ with $k \in \{0, \ldots, 4\}$. Recall that we used the same values in the experiment in Section 6.2, in which the dynamic curve was already being used. For the standard curve, we use the eight values $2^k \cdot 10^{-7}$ with $k \in \{0, \ldots, 7\}$.

We summarize the results of these experiments in two different ways (see Figure 6.12): in the first, as shown in the upper diagram, for each group and the two temperature curves, we select the value τ minimizing the average relative performance of our algorithm. The

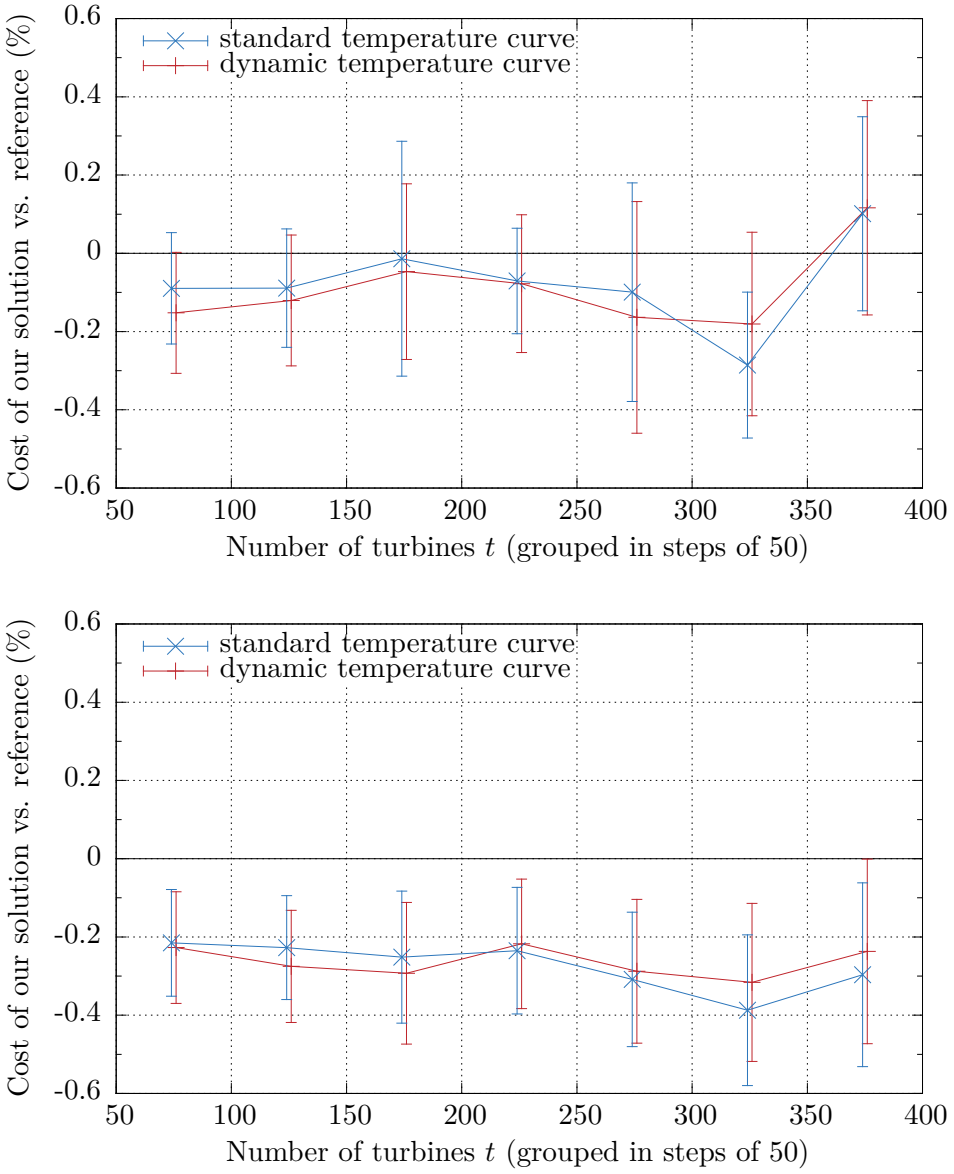


Figure 6.12: The performance of our algorithm for the instances of (T3) and (T4) for standard versus dynamic temperature curve. In the upper diagram, the best value of τ is chosen separately for each data point (i.e. group of instances) in this plot, but fixed among all instances from within that group. In the lower diagram, the best value for τ is chosen for each instance separately; accumulation of the data happens afterwards.

second one, as shown in the lower diagram, the best value of τ is chosen for each instance individually.

We draw different conclusions from these experiments. First, when comparing the results when using the standard versus the dynamic curve, the difference is almost never larger than 0.1 % in any direction, when knowing the optimal value for τ in advance. Also, it seems that the dynamic temperature curve is better suited for smaller instances (in our experiments for up to about 200 turbines), while for larger instances it seems to be disadvantageous. However, we assume that the parameter tuning for τ cannot be done for each instance individually, as this would require many executions of our algorithm for finding the perfect value of τ, before using that to actually solve the problem. Instead,

selecting a value for τ can be done with empirical knowledge, for example by solving similar-sized instances on the same machine. Then, the upper diagram from Figure 6.12 shows us that in most cases, the dynamic temperature curve has a slight advantage over the standard temperature curve also for bigger instances.

This indicates that hypothesis (H8) is true, although the improvement of the dynamic temperature curve is only very little. Also, it is only advantageous when solving smaller and medium-sized instances. We try to explain why this does not hold for larger instances: recall that one effect of the dynamic temperature curve, as seen in the upper diagram of Figure 6.11, is the compression of the early exploration phase and the extension of the late improvement phase. It seems that for larger instances, 30 minutes are already very short to properly explore the search space globally in the non-compressed exploration phase. When compressed, we might "miss" some important part of the search space, which cannot be reached in the later phase due to an already too cold temperature.

Recovering From Bad Local Optima

Our second improvement for the Simulated Annealing heuristic intends to recover from situations, in which the search space exploration is caught in a bad local optima. Hypothesis (H9) claims that the following approach improves the results.

We detect that a computation is caught in a local optimum by counting the iterations since the last time the best solution seen so far was improved. (The counter is reset to zero after such an improvement.) When the counter reaches a configurable threshold value, we assume that the current search exploration path, originating from the best solution seen so far, does not lead to a better solution. We therefore cancel that path by resetting to that best solution, i.e. we try a different search path.

With the following experiment we analyze hypothesis (H9). We also try to find the optimal iteration threshold value for the recover mechanism. For this parameter, we try the different values $2^k \cdot 10^4$ with $k \in \{0, \ldots, 6\}$. We use the instances from sets (T3) and (T4) for this experiment. To get a feeling for the dimensions of the threshold values, recall that the total number of iterations of the algorithm depends on the instance size: for 200 turbines we iterate about 8 million times, while for 500 turbines that are only about 3 million. This means that the largest parameter value already covers up to 20 % of the total running time before recovery happens, and larger values almost disable the feature effectively.

The results are compared to the solution without the recovery mechanism. (In this experiment, we do not compare against the reference solution of Gurobi, like in most other experiments.) In the plots in Figure 6.13, the vertical axis shows the relative cost when the feature is enabled: a positive value means that the recovery mechanism is disadvantageous, when using the corresponding threshold value found on the horizontal axis. A negative value indicates that the recover mechanism improves the results in average.

From these results, we draw the conclusion that the recovery mechanism should be used very carefully, if it is used at all. For the medium-sized instances from set (T3), best results are achieved when setting the iteration threshold value to $1.6 \cdot 10^5$. For larger instances, it should be set even higher. However, as explained above, such large values cover quite a great amount of the total running time of the algorithm, which effectively almost disables the feature. We therefore conclude that hypothesis (H9) is false.

As we found it very logical that this results in better solutions, we enabled this feature per default in all experiments with an iteration threshold value of 10^4. However, as discussed above, this value is too low for the recovery mechanism to be advantageous. Instead, with this value the results are worse compared to when this feature had been disabled, or used with a much larger threshold value.

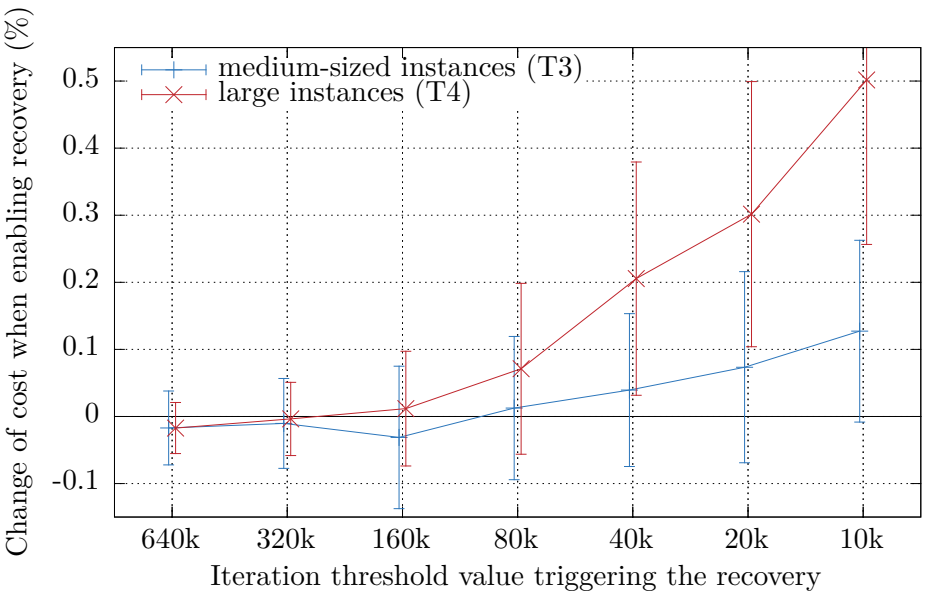


Figure 6.13: The performance of our algorithm for the instances of (T3) and (T4) with recovery mechanism being enabled versus disabled. On the horizontal axis we find different threshold values for the iteration counter. The less this value (on the right), the more quickly the recovery mechanism is activated, making it more aggressive. Note that in contrast to most other figures in this chapter, in this case the vertical axis represents the change of cost compared to not using the recovery mechanism, instead of comparing with the reference solution. Again, a lower value indicates better results.

Maintaining Multiple Instances ("Threads")

Another suggestion from Section 5.2 is to compute the Simulated Annealing procedure multiple times for the same input. We call these computations *threads*, and each one is ran with a different random seed. The idea is that, as we saw in the experiments proving (H5) on page 48, when using a different randomness, the output of the algorithm differs.

In the following experiment, we analyze if hypothesis (H10) is true. It states that in average, the best result of multiple threads is better than the result of only a single thread. The total available running time, which we again set to 30 minutes, is equally distributed among the computation threads. This means that when using too many threads, the results become worse again, as the time for each computation is then too short to result in good solutions. The goal is thus to find an optimal number of threads, which most probably depends on the instance size, as it is the major factor influencing the performance of our algorithm.

The experiment is thus performed as follows. We compare the results of using one, two and four threads for the instances from sets (T3) and (T4). Furthermore, for τ we try the two values $8 \cdot 10^{-5}$ and $16 \cdot 10^{-5}$, resulting in six experiments for each instance. We grouped the instances by their number of turbines in steps of 50. The results of the experiments are aggregated to the average and the standard deviation for each of these groups. We then decided based on that data what the best value of τ is for each instance size and number of threads. We found out that for less threads or smaller instances, $8 \cdot 10^{-5}$ is the best choice, while for more threads or larger instances, $16 \cdot 10^{-5}$ is to be preferred. We then continue with the results for the best value of τ for each experiment. We see the relative differences for the different numbers of threads in Figure 6.14.

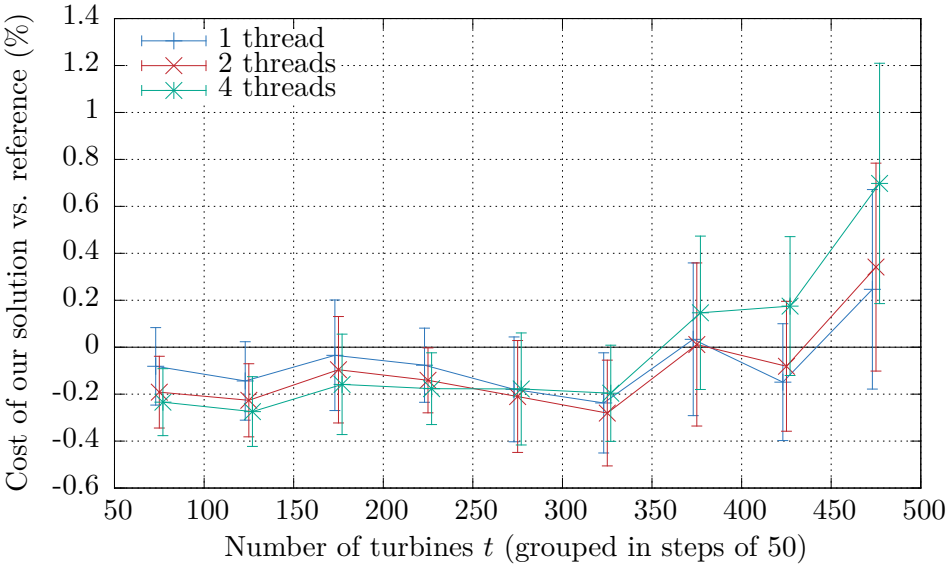


Figure 6.14: The performance of our algorithm for the instances of (T3) and (T4) when using one, two or four threads. The best value of τ is chosen separately for each data point in this plot.

We can see that for smaller instances, using more threads makes sense. However, as the instances get larger, less threads should be used. In our experiments, for instances with up to 250 turbines, the best results are averagely achieved with using four threads. For more than 250 but less than 400 turbines, two threads are the best choice, and for even larger instances only a single thread is to be preferred.

We interpret this behavior by claiming that most computations for the smaller instances reach a point at which they can not be improved further, i.e. they are probably caught in a local optimum. By trying multiple different computations, different such optima are found, among which the best is chosen by our algorithm. However, for larger instances, the computations probably do not reach that point. Instead, they suffer from the shorter running times when raising the number of threads. This results in multiple computations which cannot "finish" their search exploration properly.

Our conclusion of this experiment is that this approach makes sense for further improving already good results. It is however not suited as a speedup technique for very large instances. We can therefore confirm hypothesis (H10) for instances for which the running time is already enough in the single thread case to reach a fairly good result. We suggest our technique for a non-terminating use case, i.e. when the user does not restrict the running time but rather wants to stop the computation as soon as he is satisfied with the quality of the result.

Evolutionary Algorithm ("Crossings")

In Section 5.3, we propose to extend our algorithm with crossings, which is the idea of evolutionary algorithms. In hypothesis (H11), we claim that this technique can be used to further improve the results of multiple computations.

Recall that our suggested integration of this approach into our algorithm is as follows. The total running time (which we will set to 30 minutes like in most of our experiments) is split into the first phase, in which multiple threads are computed, like in the previous experiment. After this phase, the pair of threads which maximizes the product of diversity and compatibility is selected. They are used to construct a pair of crossings (recall that

when crossing two existing solutions results in two new solutions). Their initial temperature is set to a multiple of the average temperature of the original solutions, where the factor is a tuning parameter. In the second phase, these solutions are optimized with the same Simulated Annealing algorithm and parameter values from the first phase.

Some new parameters are introduced by this method, and we try different values for them in the following experiments. First of all, for the ratio of the total running time used for the second phase we try 10 % and 20 %. Since our total running time is 30 minutes, this means 3 or 6 minutes are used for optimizing the crossings, respectively. This shortens the first phase to the remaining 27 or 24 minutes. For the temperature factor applied to the crossings when starting the second phase, we tried the values 2, 8, 32 and 128. The idea to raise the temperature again is that the constructed representation shows up some "defects" at the border where the original representations have been cut. However, the temperature should not be set too high as that would allow the algorithm to accept much worse solutions, which would immediately destroy the information which we just extracted from the two original representations.

Recall that the above experiments showed that using four threads only makes sense for instances with up to 250 turbines (see Figure 6.14). For the parameter τ, we use the value $16 \cdot 10^{\{} - 5\}$, as this turned out to be a good parameter for wind farms of that size in previous experiments (see Figure 6.7). For the parameter τ, we use the value $16 \cdot 10^{\{} - 5\}$, as this turned out to be a good parameter for wind farms of that size in previous experiments (see Figure 6.7). We decided to focus on the instances of set (T3) for this experiment.

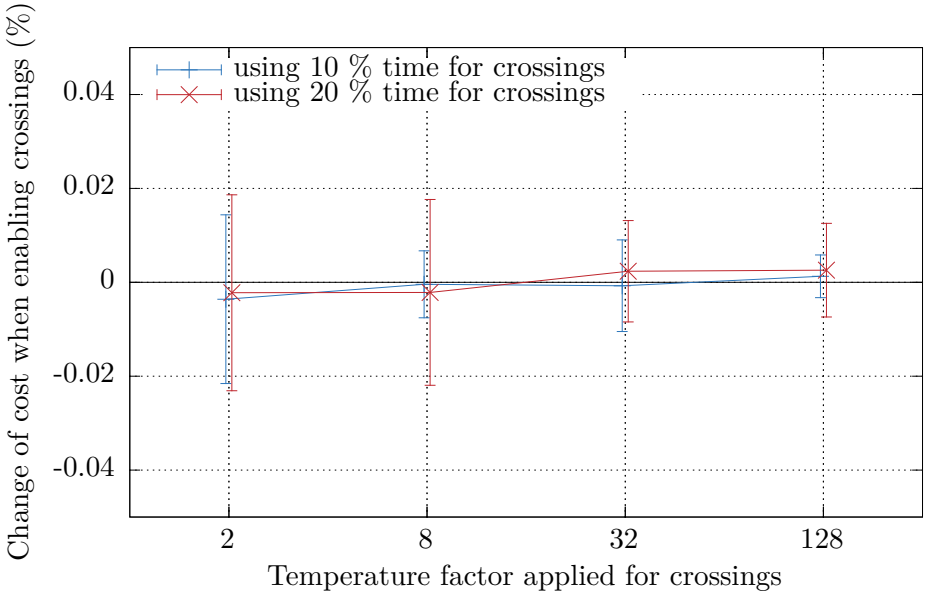
In Figure 6.15, we compare the cost of the solutions for the eight different combinations of parameters to the solution when no crossings are enabled at all. In contrast to the previous experiments, this plot shows on the vertical axis the relative cost of the solution with crossings compared to the solution without crossings. Again, smaller values indicate better results. In particular, a positive value means that crossings are disadvantageous for that particular choice of parameters.

For the smaller instances with up to 120 turbines, we notice that the difference is only very tiny: even for the best choice of parameters, the crossings improve the result not even by 0.01 % in average. When considering the standard deviation of each data point, we see that for some instances, the crossings are advantageous, while for some other instances, it is disadvantageous. However, as we are talking about very small percentages, this effect can be ascribed to statistical inaccuracy.

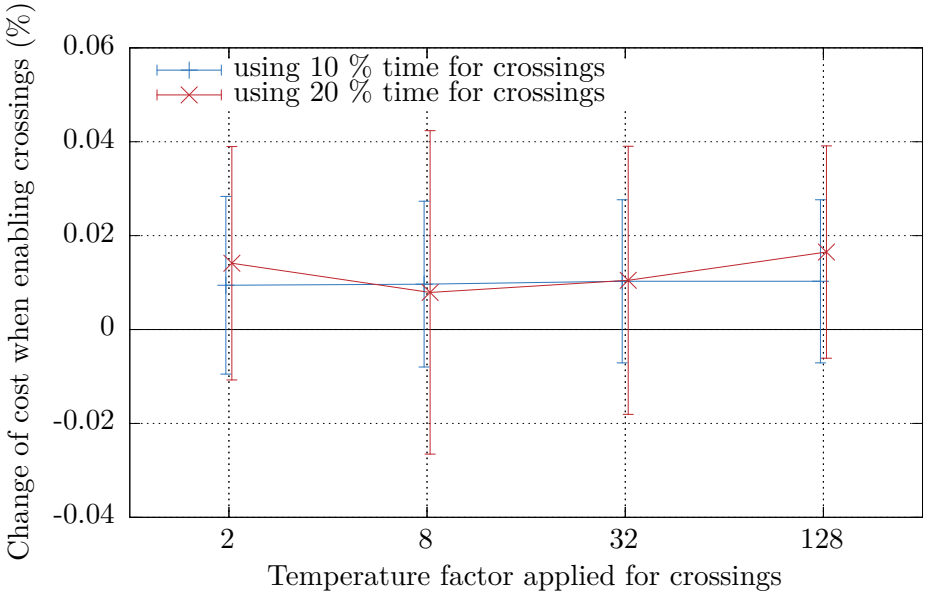
For most of the instances with 120 to 200 turbines, enabling crossings seems to be disadvantageous. Again, for some instances we gained better results, but here the average change of cost suggests not to use crossings.

We conclude that this suggested method is not an improvement to our heuristic in most cases, disproving hypothesis (H11). However, this does not mean that the idea of incorporating crossings into our Simulated Annealing-based heuristic is bad in general.

It is difficult to explain why the suggested method does not work. We guess that the most difficult part when incorporating crossings into our heuristic is how these are constructed. In our case, the idea of cutting two representations in halves, crossing these over and concatenating them to full representations seems plausible at first. However, it seems to be that this leaves some defects at the cutting border, which are difficult to fix. In future work, a different method for crossing two representations could be engineered. We think that in general, the crossing approach could be an improvement to the Simulated Annealing heuristic.



(a) Instances with 80 to 120 turbines



(b) Instances with 120 to 200 turbines

Figure 6.15: The change of cost of the globally best solution when using crossings, compared to not using crossings. In these experiments we use four threads. The instances are from (T3), which we split into two subsets with 80 to 120 (a) and 120 to 200 turbines (b). For the last 10 or 20 % of the available running time (which is again 30 minutes) one pair of crossing is computed. The initial temperature of these two threads is set to a multiple of the average temperature of the original threads. We try different factors for this. Note that in contrast to most other figures in this chapter, in this case the vertical axis represents the change of cost compared to not using crossings (but still four threads), instead of comparing with the reference solution. Again, a lower value indicates better results.

7. Conclusion

In the experiments from Chapter 6, we have seen that our heuristic is a good alternative to the MILP model for small and medium-sized wind farms. For our experimental instances with up to 450 turbines, our solution outperforms the results of Gurobi, even after a shorter amount of time. However, we see a couple of problems with our approach.

We had problems solving large wind farms with more than 450 turbines. An explanation could be that the representation structure we use in Simulated Annealing is not suited for such a large graph size, and an alternative representation should be used instead. Also, when the capacities of the substations are very tight, i.e. almost equal the total amount of power of the whole wind farm, our algorithm has difficulties with assigning the turbines to the substations. We aimed to solve both problems with the two-level approach, which first partitions the wind farm into substation networks, and then optimizing the cabling for each substation separately. However, we had difficulties with the partitioning step.

Furthermore, tuning some parameters, such as τ, the velocity of the temperature drop, seems to be very critical for the algorithm's performance. However, we found it difficult to find a good strategy to choose an optimal parameter value in advance.

Improving our Heuristic

We therefore suggest that our algorithm can be improved in future work with the following ideas.

First of all, a different representation could be used in the Simulated Annealing framework. For this, we primarily think of the intermediate representations we explained in Figure 5.1 on page 24. Furthermore, we complement the list of alternative representations by the following two ideas. The first restricts the set of edges to only trees. This is similar to our edge cuts but with a fixed cardinality such that just enough edges to form one tree per substation are remaining. The mutation would then just swap the states of an enabled and a disabled edge. The second alternative idea assigns each edge its maximal cable type (but does not say anything about whether that cable is actually used on that edge). Edges near a substation tend to use thicker cables, while edges far away only use smaller ones. This serves as a capacity constraint, and we evaluate that by optimizing the resulting CMST (for example with the Esau-Williams heuristic). The mutation would lift or reduce the cable type on an edge.

For the two-level approach, a different substation assignment construction could be used. As we saw in Section 5.4, the method we use to split a solution into individual substation

sub-problems does not work very well. Yet, the general idea of splitting the problem into the tasks of assigning each turbine to a substation and optimizing the single substation cablings seems to be very promising. We conclude this from our experiments in which we found out that the main complexity of the cabling problem arises from the first of these two tasks. An alternative method to construct the sub-problems could be to primarily focus on how the power is routed in the final solution, instead of to which substation each turbine path is routed.

A hybrid approach which unifies Simulated Annealing and Linear Programming could also be interesting. Here, different kinds of hybrid solutions seem to be plausible: we could first use an MILP to find a good initial solution which is then fine-tuned using our heuristic. More concretely, this could be implemented using a two-level approach similar to the one we proposed in Section 5.4. A different option could be to optimize a problem instance using our Simulated Annealing heuristic, but to not use the resulting cabling, but rather assign the edges a weight depending on how often the heuristic used that edge. Given this weight, we can construct hardening constraints for the MILP (which is again a heuristic approach similar to how we restrict the edges in our experiments).

Summarizing, we think our Simulated Annealing-based heuristic serves as a good basis for developing more complex algorithms. In particular, we think that one of the different kinds of two-level approaches can be a good alternative for the linear programming approach to optimize the cabling for even larger wind farms.

Other Optimization Problems in Wind Farms

We only covered a particular optimization problem when designing wind farms: finding a cable layout with minimal installation cost. We assume fixed locations for all turbines and substations. We already suggest an MILP model to find optimal locations for the substations, such that the cabling becomes cheapest. We did not have enough time to integrate this into our Simulated Annealing heuristic, however we think that this could be tried.

Besides that, we find the following other optimization problems interesting, which could be covered in future work.

In addition to not fixing the substation locations, we could optimize their quantity and capacity from a given set of substation types, similar to how we choose between different cable types for each edge.

When constraining the locations of the turbines and substations in the input, such as defining their minimum pairwise distance, the problem could become more easily to be solved. There might be an approximation scheme making use of this additional information.

Furthermore, in our problem model, only at turbines and substations cables are allowed to be merged. However, when relaxing this to allow arbitrary merge points, the cabling cost could be reduced, although the added point might imply additional costs. This is similar to the Steiner Tree problem, which is also NP-hard [HR92]. Specifically for the design of electrical circuits, there is the Transmission Network Expansion Planning (TNEP) problem, which can be solved using Genetic Algorithms [dSGA99].

We could furthermore forbid that cables cross each other, as this requires to bury one of the two cables deeper than the other to avoid conductive influences. Instead of forbidding, we can also charge an additional cost for an intersection. We believe that this could be incorporated into the Simulated Annealing heuristic.

Bibliography

[4coa] Website: BARD Offshore 1 project data – C4 Offshore. http://www.4coffshore.com/windfarms/windfarms.aspx?windfarmId=DE23, Accessed: 2016-05-04.

[4cob] Website: BorWin 1 Converter – C4 Offshore. http://www.4coffshore.com/windfarms/hvdc-converter-borwin1-converter-cid1.html, Accessed: 2016-05-04.

[4coc] Website: Global Offshore Wind Farm Map – C4 Offshore. http://www.4coffshore.com/offshorewind/, Accessed: 2016-05-04.

[bar] Website of the BARD Offshore 1 project – Ocean Breeze. http://www.oceanbreeze.de/, Accessed: 2016-05-09.

[Ber] Constantin Berzan. Algorithms for Cable Network Design on Large-scale Wind Farms. http://thirld.com/files/msrp_techreport.pdf, Accessed: 2016-01-04.

[CGJ+] M. Chimani, C. Gutwenger, M. Jünger, G. W. Klau, K. Klein, and P. Mutzel. The Open Graph Drawing Framework (OGDF). Website: http://www.ogdf.net/, Accessed: 2016-05-11.

[cpl] Website of the IBM ILOG CPLEX Optimizer Software. http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/, Accessed: 2016-05-04.

[Dor01] Marco Dorigo. *Advances in Artificial Life: 6th European Conference, ECAL 2001 Prague, Czech Republic, September 10–14, 2001 Proceedings*, chapter Ant Algorithms Solve Difficult Optimization Problems, pages 11–22. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[dSGA99] E. L. da Silva, H. A. Gil, and J. M. Areiza. Transmission network expansion planning under an improved genetic algorithm. In *Power Industry Computer Applications, 1999. PICA '99. Proceedings of the 21st 1999 IEEE International Conference*, pages 315–321, Jul 1999.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness.* W. H. Freeman and Company, 1979.

[GL99] Fred Glover and Manuel Laguna. *Handbook of Combinatorial Optimization: Volume1–3*, chapter Tabu Search, pages 2093–2229. Springer US, Boston, MA, 1999.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

[gur] Website of the Gurobi Optimizer Software by Gurobi Optimization Inc. http://www.gurobi.com, Accessed: 2016-05-04.

[HCB06] P. D. Hopewell, F. Castro, and D. I. Bailey. Optimising the design of offshore wind farm collection networks. In *Proceedings of the 41st International Universities Power Engineering Conference*, volume 1, pages 84–88, Sept 2006.

[HR92] F. K. Hwang and Dana S. Richards. Steiner tree problems. *Networks*, 22(1):55–89, 1992.

[JR04] Raja Jothi and Balaji Raghavachari. Revisiting esau-williams' algorithm: On the design of local access networks. In *IN PROC. 7TH INFORMS TELECOMMUNICATIONS CONF*, pages 104–107, 2004.

[JR05] Raja Jothi and Balaji Raghavachari. Approximation algorithms for the capacitated minimum spanning tree problem and its variants in network design. *ACM Trans. Algorithms*, 1(2):265–282, October 2005.

[lp-] lp-solve. A Mixed Integer Linear Programming (MILP) solver, free under the LGPL, http://lpsolve.sourceforge.net/.

[Min86] M. Minoux. Solving integer minimum cost flows with separable convex cost objective polynomially. In *Mathematical Programming Studies*, volume 26 of *Lecture Notes in Computer Science*, pages 237–239. Springer Berlin Heidelberg, 1986.

[OL96] Ibrahim H. Osman and Gilbert Laporte. Metaheuristics: A bibliography. *Annals of Operations Research*, 63(5):511–623, 1996.

[ope] Website of the Openwind software tool – AWS Truepower Software. `http://software.awstruepower.com/openwind/`, Accessed: 2016-05-02.

[qt] The Qt Company. Qt Framework. `http://www.qt.io`, Accessed: 2016-05-11.

[SW97] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *Journal of the ACM*, 44(4):585–591, July 1997.

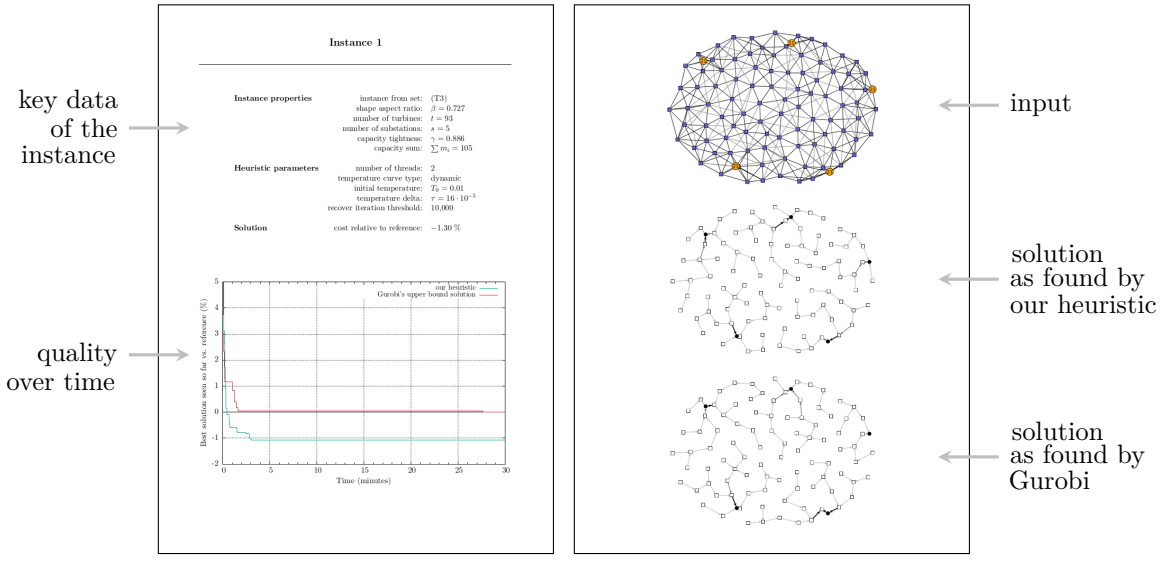
[YA12] Masoud Yaghini and Mohammad Rahim Akhavan. A simulated annealing algorithm for unsplittable capacitated network design. *International Journal of Industrial Engineering & Production Research*, 23:91–100, June 2012.

Appendix

On the following pages, we show some of our instances we used in our experiments, together with the results of the MILP as well as our heuristic. For this, we selected two instances from (T3) and one from (T4). Each such instance makes up a double page.

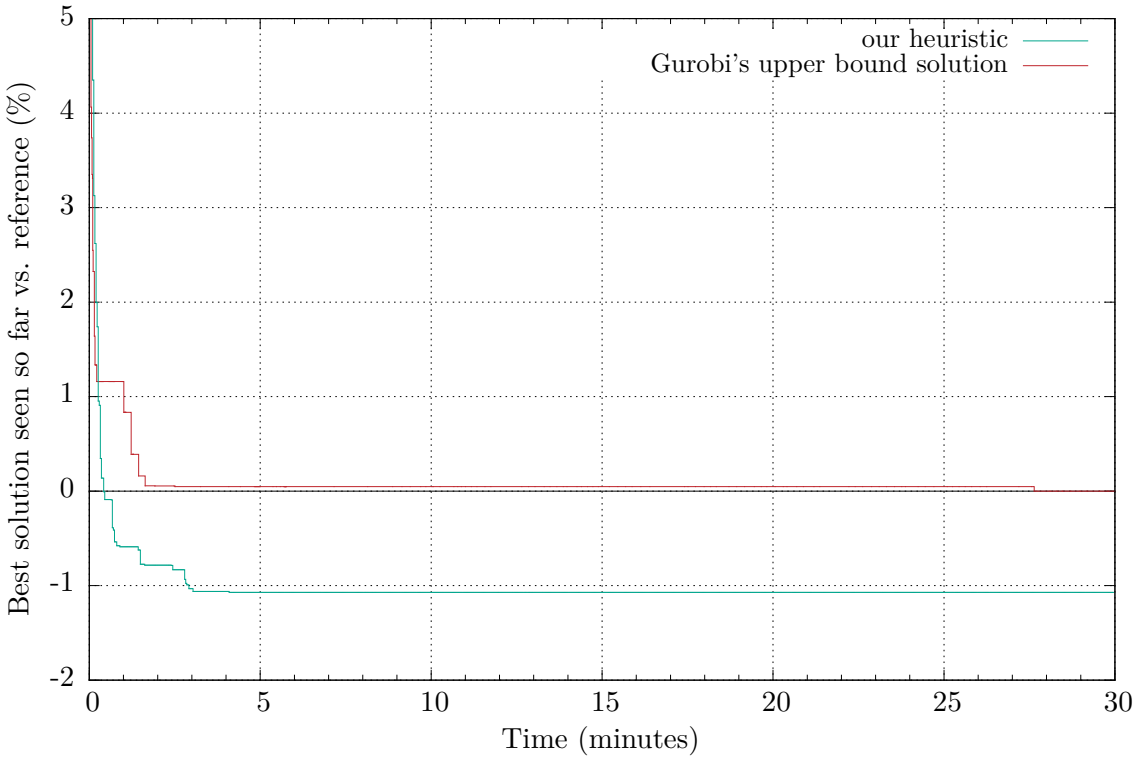
On each left page, a table shows the parameters used to generate the instance, as well as the parameters used for our Simulated Annealing heuristic. Below, a diagram shows the cost of the best solution seen so far over time, relative to the reference solution. A second curve shows the cost of the corresponding solution of Gurobi Optimizer. Note that we continue Gurobi for another half an hour, resulting in the reference solution. However, the curve for the second half an hour is not displayed in this diagram.

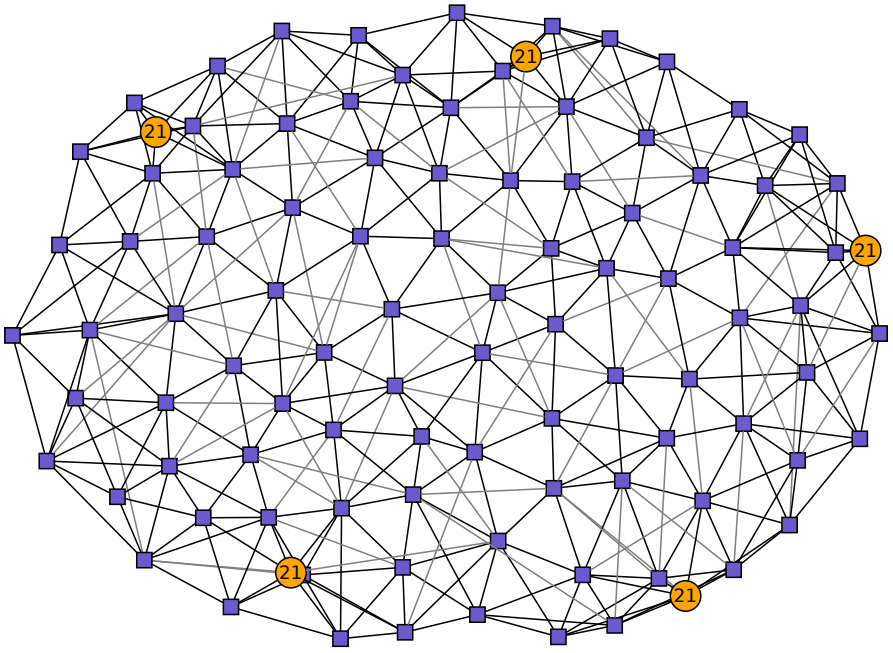
The right page shows three graphs of which the first one is the input. Below is the result of our algorithm with the best choice of tuning parameters we tried throughout all experiments. The bottom figure shows the final upper bound solution of Gurobi Optimizer solving the MILP with a time limit of one hour, which is our reference solution. In the graphs showing the results, the chosen cable type of an edge is visualized with different thicknesses. The first three cable types are furthermore drawn with different shades of gray, and the fourth is drawn in blue color (only appearing in our result for the third instance).

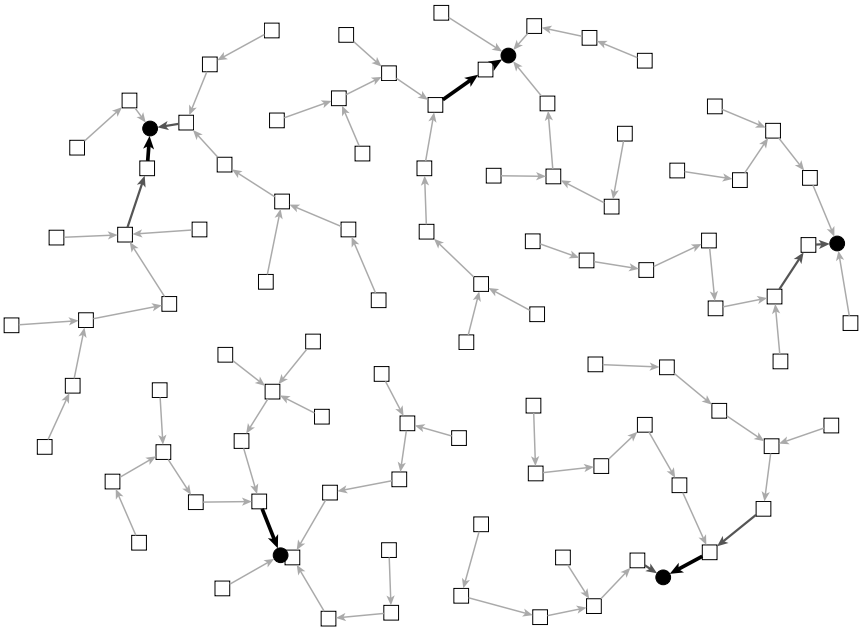


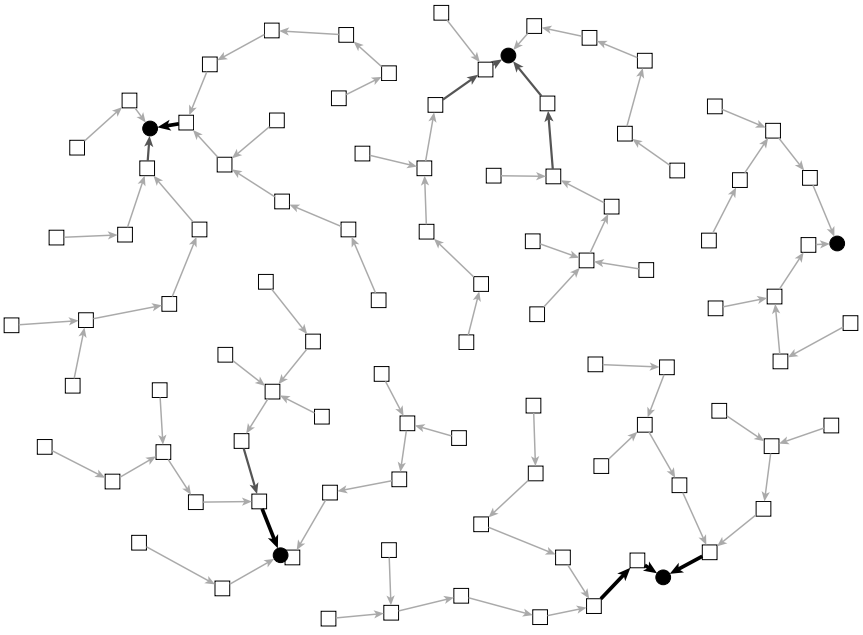
Instance 1

Instance properties	instance from set:	(T3)
	shape aspect ratio:	$\beta = 0.727$
	number of turbines:	$t = 93$
	number of substations:	$s = 5$
	capacity tightness:	$\gamma = 0.886$
	capacity sum:	$\sum m_i = 105$
Heuristic parameters	number of threads:	2
	temperature curve type:	dynamic
	initial temperature:	$T_0 = 0.01$
	temperature delta:	$\tau = 16 \cdot 10^{-5}$
	recover iteration threshold:	10,000
Solution	cost relative to reference:	-1.30 %



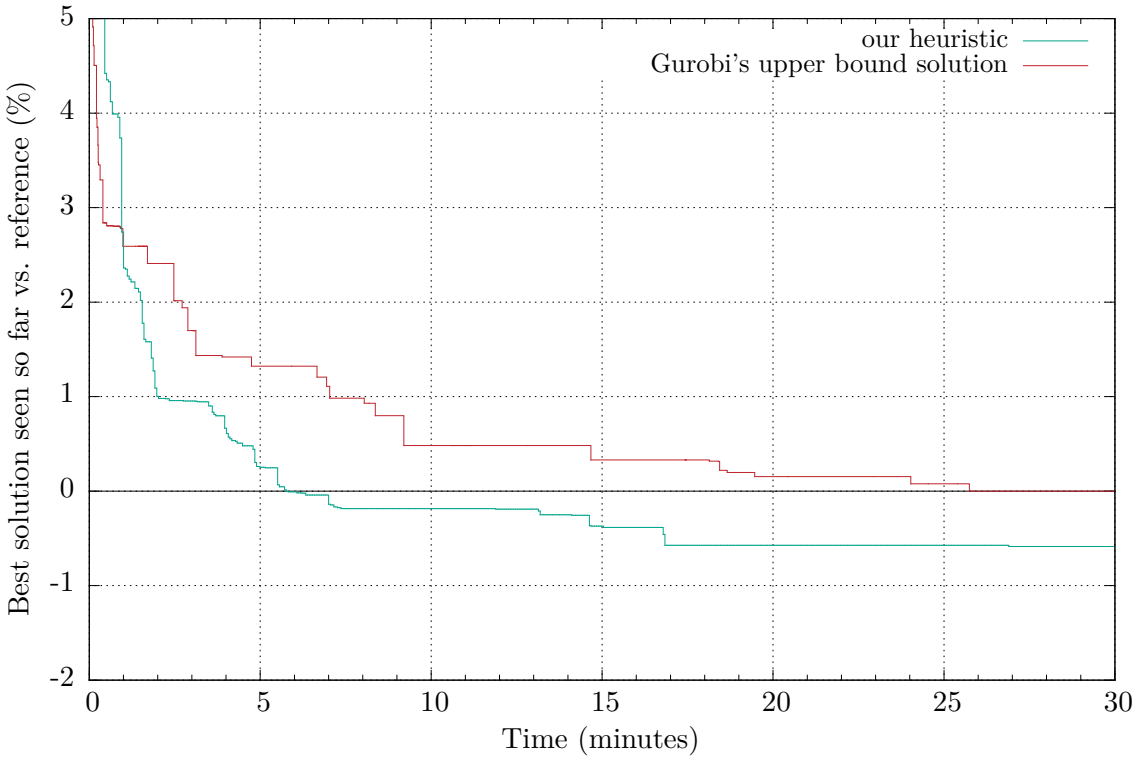

21
21
21
21
21

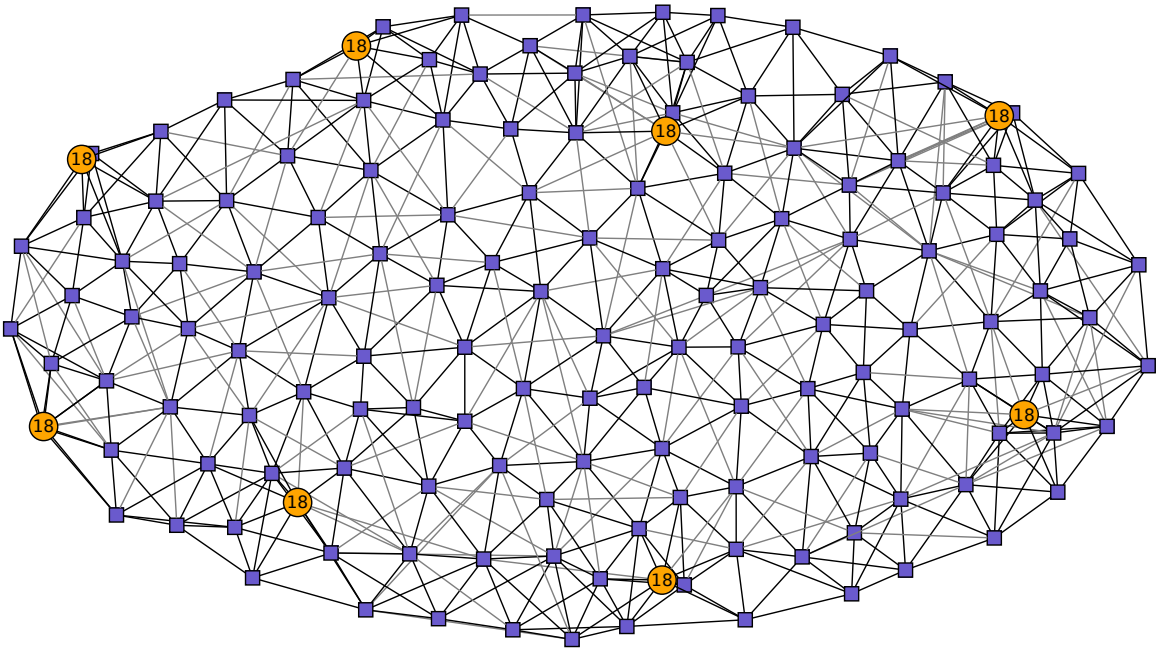


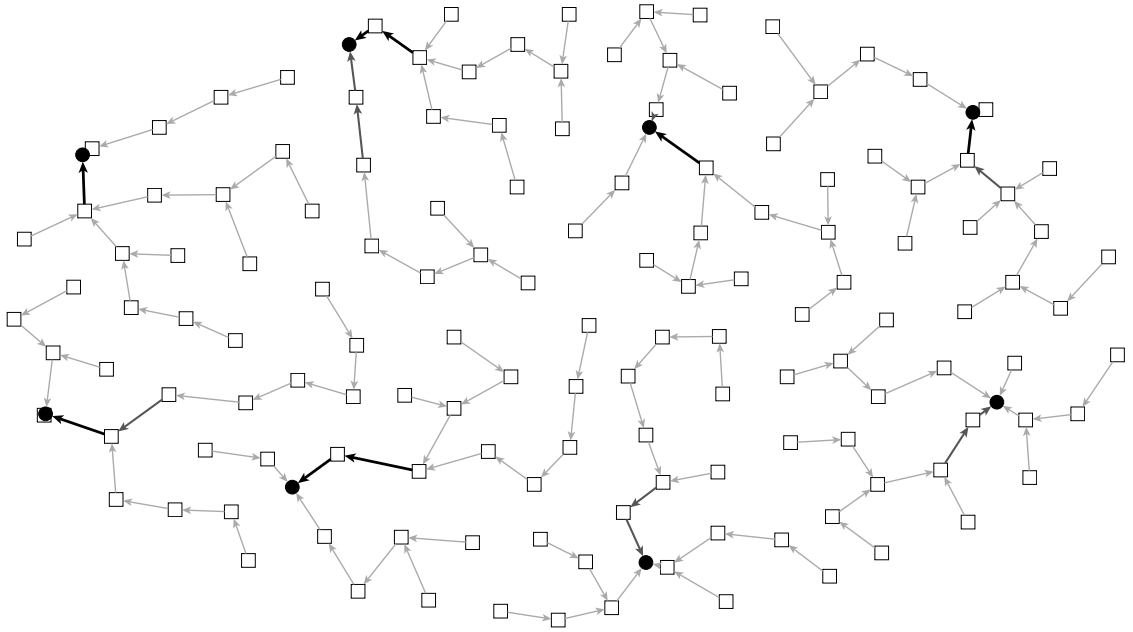


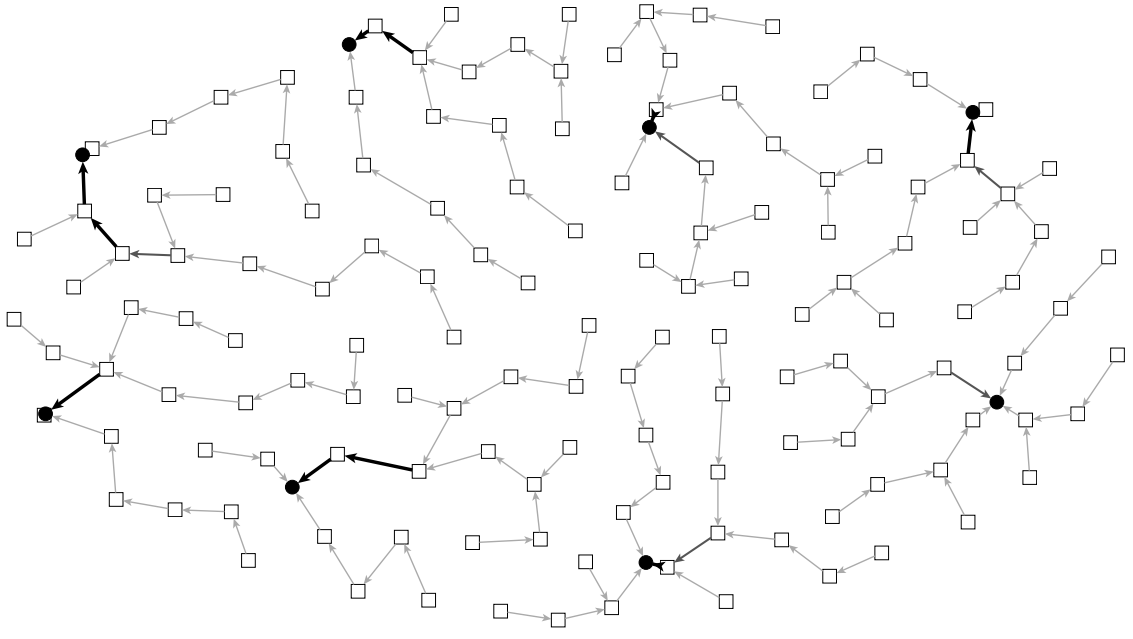
Instance 2

Instance properties	instance from set:	(T3)
	shape aspect ratio:	$\beta = 0.558$
	number of turbines:	$t = 140$
	number of substations:	$s = 8$
	capacity tightness:	$\gamma = 0.972$
	capacity sum:	$\sum m_i = 144$
Heuristic parameters	number of threads:	1
	temperature curve type:	dynamic
	initial temperature:	$T_0 = 0.01$
	temperature delta:	$\tau = 8 \cdot 10^{-5}$
	recover iteration threshold:	10,000
Solution	cost relative to reference:	-0.59 %



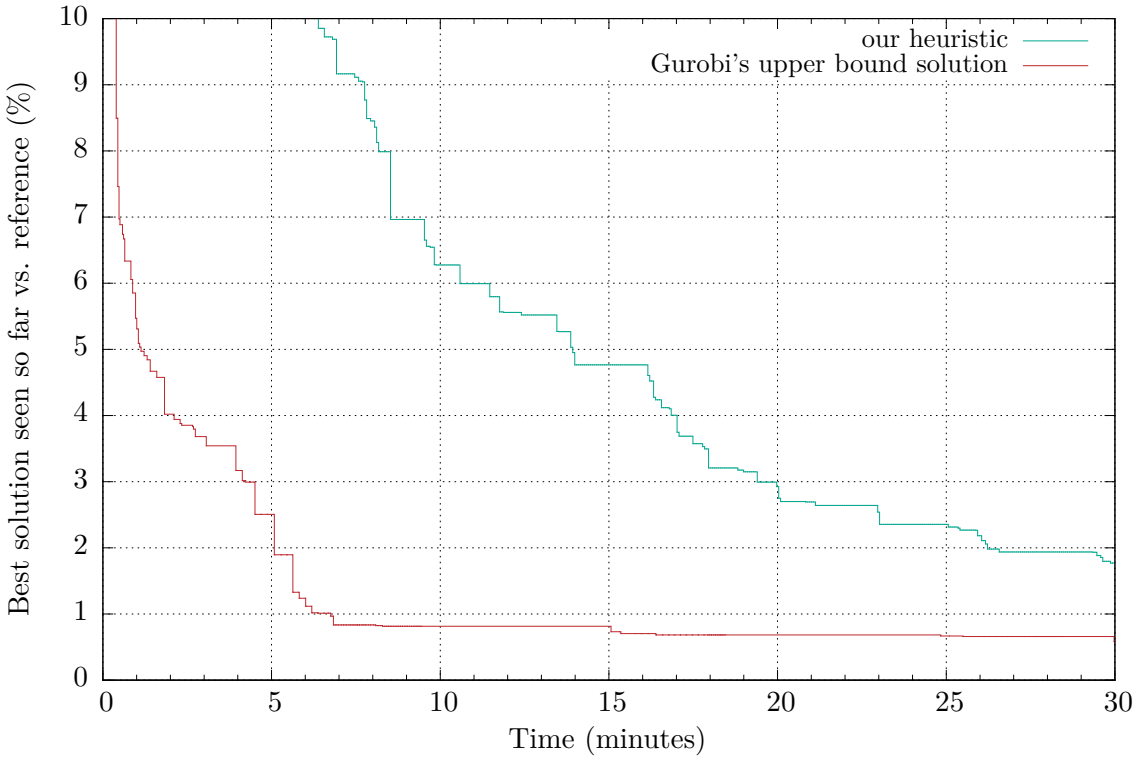

18
18
18
18
18
18
18
18

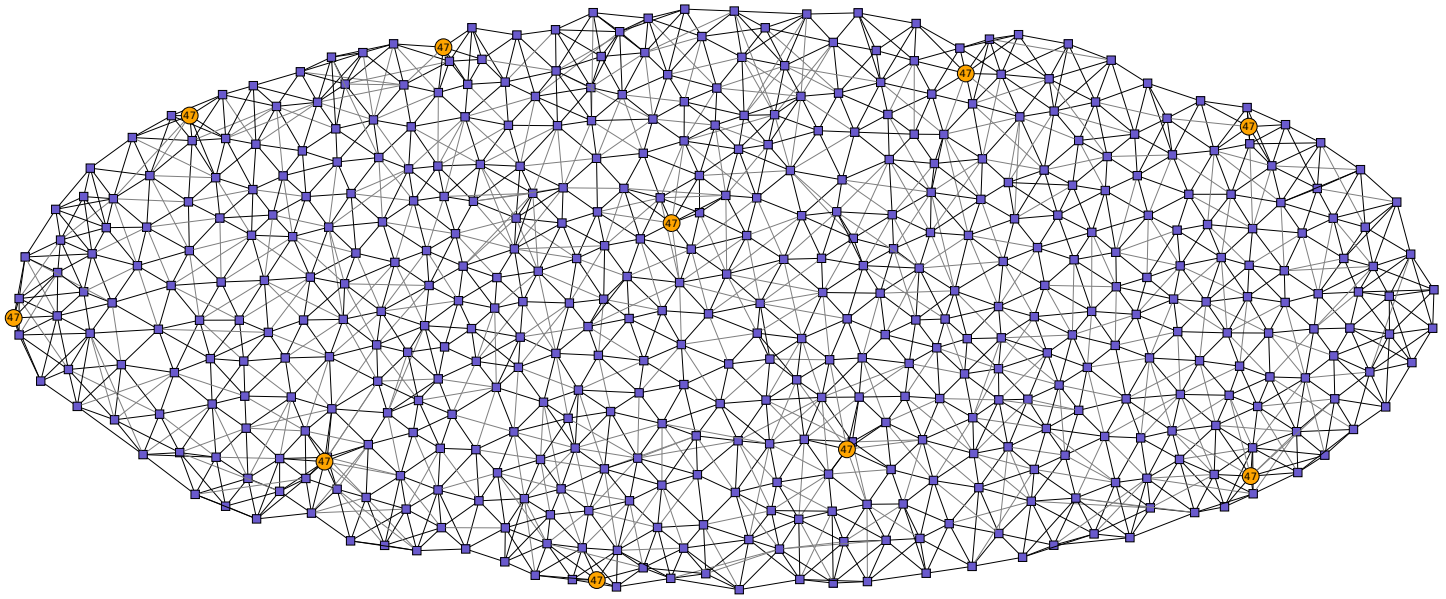




Instance 3

Instance properties	instance from set:	(T4)
	shape aspect ratio:	$\beta = 0.409$
	number of turbines:	$t = 463$
	number of substations:	$s = 10$
	capacity tightness:	$\gamma = 0.985$
	capacity sum:	$\sum m_i = 470$
Heuristic parameters	number of threads:	1
	temperature curve type:	dynamic
	initial temperature:	$T_0 = 0.01$
	temperature delta:	$\tau = 8 \cdot 10^{-5}$
	recover iteration threshold:	∞ (disabled)
Solution	cost relative to reference:	-1.74 %




47
47
47
47
47
47
47
47
47
47
47