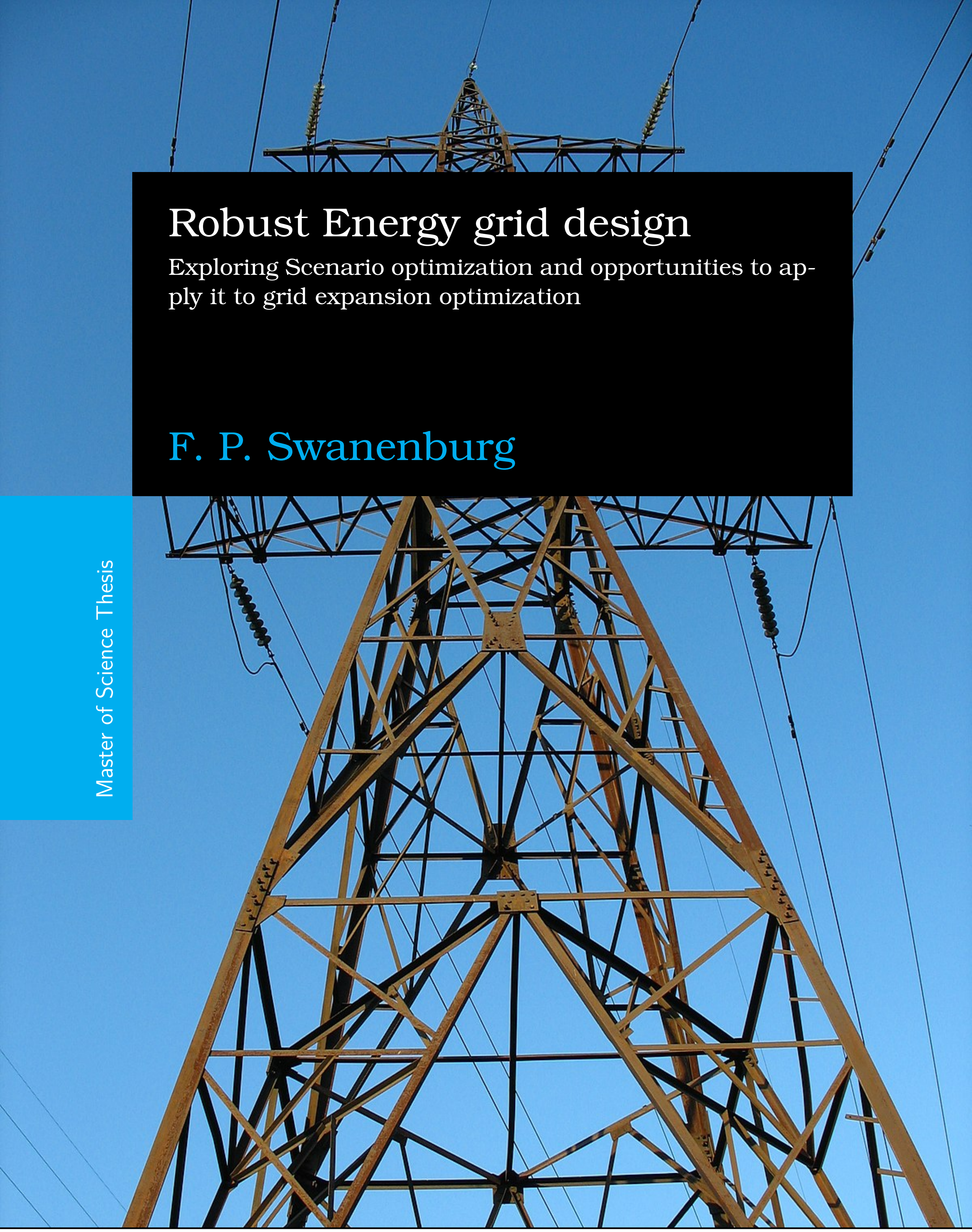

Robust Energy grid design

Exploring Scenario optimization and opportunities to apply it to grid expansion optimization

F. P. Swanenburg

Master of Science Thesis

Delft University of Technology

Delft Center for Systems and Control

Robust Energy grid design

Exploring Scenario optimization and opportunities to apply it to grid expansion optimization

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

F. P. Swanenburg

April 30, 2025

Faculty of Mechanical Engineering (ME) · Delft University of Technology

Delft University of Technology
Department of
Delft Center for Systems and Control (dcsc)

The undersigned hereby certify that they have read and recommend to the Faculty of Mechanical Engineering (ME) for acceptance a thesis entitled

Robust Energy grid design

by

F. P. Swanenburg

in partial fulfillment of the requirements for the degree of

Master of Science Systems and Control

Dated: April 30, 2025

Supervisor(s): ______________________
Dr. G. Pantazis

____
Dr. S. Grammatico

Reader(s): ______________________
Dr. P. Mohajerin Esfahani

Abstract

This thesis aims to apply the scenario optimization method to grid expansion, which aims to improve operations and reliability of a grid. Thus far, a large part of grid expansion research has used Monte-Carlo optimization methods in finding the best improvements for the grid. This thesis aims to explore the possibility of using a different optimization model.

The methodology developed in this thesis aims to leverage the robustness claims made by scenario optimization to achieve better expansion results than the Monte-Carlo approach to grid expansion optimization. To do so, three optimization models are developed with the goal of comparing two new scenario methods like-for-like with the prevalent Monte-Carlo method.

Four case studies show that the scenario approach does achieve comparable to or better results than the Monte-Carlo approach, and do so in considerably less time. These simulation studies show that the scenario approach might be a suitable alternative to the currently used methods.

Secondly, a significant improvement in terms of scenario optimization operational performance is associated with a comparable relavite improvement in Monte-Carlo optimization operational performance, allowing furhter grid expansion studies to make informed decisions on which grid modifications to fully study.

Keywords: Grid expansion, Scenario approach, Robust optimization, Monte-Carlo optimization, Optimal Power Flow, Grid operation, Grid reliability

Acknowledgements

I would like to express my sincere gratitude to the following individuals who have played a significant role in the successful completion of my master's thesis.

First of all, I would like to thank Giorgos for his supervision and guidance. The insightful suggestions and discussions have been a major contribution to this thesis and have made it a very enjoyable process.

I would like to express my appreciation to my housemates for their understanding, motivation, and support. Finally, I am thankful for my parents and sisters as their support and belief in me have been crucial not only during this thesis, but my entire academic journey.

Delft, University of Technology
April 30, 2025

F. P. Swanenburg

Table of Contents

List of Symbols

Scenario optimization

θ	Design parameter
Θ	Domain of design parameter θ
n_θ	Size of design parameter θ
δ	Uncertainty parameter
Δ	Domain of uncertainty parameter δ
n_δ	Size of of uncertainty parameter δ
Θ_i^δ	Constraint on θ, based on uncertainty parameter δ
Θ^δ	Union of all constraint on θ, based on uncertainty parameter δ
$\Theta_i^{(\delta_j)}$	Constraint on θ, based on sample δ_j of uncertainty parameter δ
$\Theta^{(\delta_j)}$	Union of all constraints on θ, based on sample δ_j of uncertainty parameter δ
ϵ	Level parameter
β	Confidence parameter
$V(\theta)$	Violation probability
$f_{V(\theta)}$	Probability density function of violation probability $V(\theta)$
$\hat{\theta}_N$	Optimal value of design parameter θ give samples N
N	Number of samples / scenarios
k	Complexity; Number of support constraints
K	Set of support constraints
R	Number of discarded scenarios
$I_{discarded}$	Set of discarded scenarios

Optimal power flow

$\mathcal{G}$	Graph
$\mathcal{V}$	Set of nodes in graph $\mathcal{G}$
$\mathcal{E}$	Set of edges in graph $\mathcal{G}$
$\mathcal{W}$	Set of admittances of lines in set $\mathcal{E}$ in graph $\mathcal{G}$
$w_{i,j}$	Admittances of line between node i and j
$w_{sh,i}$	Shunt admittance at node i
n	Number of nodes in graph $\mathcal{G}$
Y	Admittance matrix
Z_p, Z_q	Impedance matrix of active- and reactive power
P_δ, Q_δ	Uncertain, active- and reactive power loads
P_θ, Q_θ	Controlled, active- and reactive power power loads
V_0	Baseload voltage level
V_{min}, V_{max}	Vector of upper and lower bounds on voltage level
P_{min}, P_{max}	Vector of upper and lower bounds on controlled active power
Q_{min}, Q_{max}	Vector of upper and lower bounds on controlled reactive power

Grid expansion

h_{pdf}	Probability that one violation probability is lower than another, using probability density functions
H_{pdf}	Matrix with elements h_{pdf}
h_{post}	Probability that one violation probability is lower than another, using a-posteriori ϵ values
H_{post}	Matrix with elements h_{post}
D	Branch depth, development horizon
B	Branch breadth, exploration variable for longer development horizon
h	Current exploration depth in step of optimization loop
N	number of optimal power flow computations necessary for grid expansion step
$f(S^*)$	Operational performance cost function
$\hat{f}_{\mathcal{G}}^{Sc}$	Optimal operational value when using Scenario optimization
$\hat{f}_{\mathcal{G}}^{MC}$	Optimal operational value when using Monte-Carlo optimization
Z^*	Extended impedance matrix
$\delta^{(i)}$	Uncertainty sample, sampled power load
S^*	Controlled power load
U_{add}	Set of possible cable additions
w	Capacity of added power line
U_{upg}	Set of possible cable upgrades
w^+	Capacity upgrade for upgrading power lines
U	Set of all possible modifications
U_{branch}	Sequence of modifications investigated as a combination of modifications
$U_{branch}^{i,j,\ldots,k}$	Sequence of modifications investigated as a combination of modifications, indexed by path chosen
T_D^B	Tree set of all branch sequences $U_{branch}^{i,j,\ldots,k}$, denoting the sub-tree explored by the collective of these branches
$\hat{U}_{branch}$	(Locally) Optimal sequence of modifications investigated as a combination of modifications
$\hat{u}$	Optimal graph modification
$\hat{U}$	Sequence of previously optimal modifications, currently installed
g_1^{Sc}	Grid expansion cost function, using the scenario approach with horizon 1
g_1^{MC}	Grid expansion cost function, using the Monte-Carlo approach with horizon 1
g_h^{Sc}	Grid expansion cost function, using the scenario approach with horizon h
W	Weight vector for grid expansion
$d(u)$	Normalized length of addition u
c_{mod}, c_{tot}	Stopping criteria on optimization loop

Validation stage

m	Number of repeats for empirical violation probability study
N_{MC}	Sample size used during Monte-Carlo stage of result validation
N_{novel}	Number of novel sample for each empirical violation probability study

Chapter 1

Introduction

1-1 Background

The EU aims to increase the fraction of energy we use that is green energy, as a means to evade climate catastrophe. With a lot of green energy sources, however, grid operators run into the problem of intermittency; They cannot control when the power source does or does not provide the grid with power. This has put a lot of pressure on grid operators to improve their grid design in order to still attain the same reliability standards that they have done before.

Provided that there is enough fuel, coal and gas power plants can be kept running indefinitely. They can provide a baseload, or be kept on standby to be dispatched to meet energy demand.

In contrast, renewable power plants can only generate electricity when the conditions are right; Solar power needs the sun to shine, and wind power needs the wind to blow the right amount. The intermittency challenge of these renewable sources makes it harder to meet the demand, and is a challenge inescapable with increasing renewable energy supply.

With that intermittency also comes the issue that most of these renewable energy sources are concurrent; When the sun shines, all solar panels in the area will deliver a heightened load simultaneously, or when the wind blows, all wind farms will supply more power. This leads to increased load peaks on the transmission and distribution grids.

This has already led to grid operators in the Netherlands to stop connecting solar farms in areas with less demand. And large energy users are also incidentally refused their connection. The power grid simply cannot efficiently cope with the daily peaks of electric power. [1, 2]

A second issue is that, because the power supply is no longer controlled by any operator, the daily cycle and yearly cycles of solar and wind energy production does not coincide with the energy use. The distribution problem is no not only spatial, i.e. transmitting it from *the location of* supply to *the location of* use, but also temporal, i.e. transmitting it from *time of* supply to *time of* use. [3]

Currently, the grid development plans in Europe are mostly not ambitious enough to cope with the renewable energy supply expansion plans. [4]

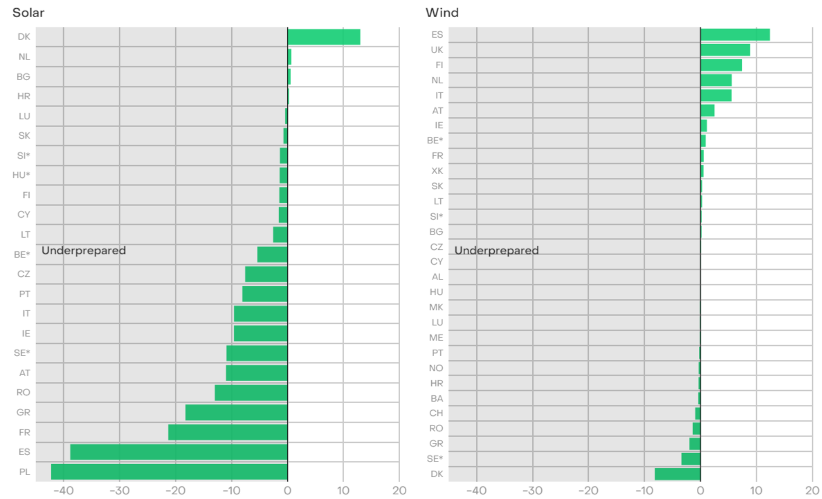

Figure 1-1: The expansion plans of various European countries compared to their solar and wind power ambitions. [4]

But there is some upturn. The Dutch government has provided TenneT, the Transmission Grid Operator of the Netherlands, with a loan of 25 Billion Euros in order to invest in upgrading the grid as soon as possible. [5]

1-2 Related work

There is a consistent research effort on grid expansion, making sure that the correct expansion choices are made. Up to this point, reliability considerations in grid expansion research have mostly been a worst-case approach or making use of some index. The objective has generally been to provide electricity as cheaply as possible, with improved reliability only being part of that. [6, 7]

One method of optimization respecting reliability, is scenario optimization. Over the years, research has been done both on increasing the value of the result of scenario optimization, as well as applying scenario optimization to grid operation. Among other things, efforts have led to a decrease in the number of samples required to conclude some reliability, or make more efficient use of the allowed violation probability. [8, 9]

1-3 Problem statement

The goal of this thesis is to develop a method of applying scenario optimization to grid expansion, exploiting the robust nature of this optimization technique. The main research aim is to show that results acquired when applying the scenario approach to grid expansion hold up with the results acquired when using the Monte-Carlo approach. The second research question is if using the information given on reliability by scenario optimization explicitly in the expansion optimization yields better results. Lastly, we aim to find all drawbacks or advantages of the scenario approach that can be used to achieve even better results.

To realize these goals, a power flow model will be chosen, and three optimization models developed; Using the Monte-Carlo approach, the scenario approach and the scenario approach for different development horizons. These models are developed in such a way that the model architecture was consistent between the three. Using these optimization models, three parameter studies are run comparing optimization models, and optimizing with or without explicitly using robustness information from the scenario optimization approach.

The current grid expansion research is laid out in section 2-1. The scenario approach is introduced in section 2-2. The power flow model is described in section 2-3. The optimization model is designed in section 3-1, and the result validation stage is described in section 3-2. Section 3-3 lays out all parameter studies run, and section 4 contains a selection of the results of those parameter studies. Finally, based on the simulation studies a conclusion is drawn in section 5. Furthermore, a discussion is provided in section 6 and potential future work will be discussed as well.

Chapter 2

Theory

2-1 Grid expansion

In grid expansion literature, most studies use a optimization scheme on a single sample set to calculate optimal power flow, and then optimize over an average of those power flow models to optimize the grid by grid expansion. This effectively is a Monte-Carlo approach to grid expansion.

Reliability is often considered as an a-posteriori resulting index and checked, and if it is included in the actual optimization it is often considered a financial liability or full information on the grid is assumed to be known.

Some papers have expanded research into heuristic methods of grid expansion, trying to improve on the combinatorial nature of the problem, whilst others have developed different approaches to simplifying either the power flow problem or the grid expansion problem into smaller subproblems to be solved independently or in tandem.

2-1-1 Decision variables

In survey study [6], the decision variables used in grid expansion studies are described. These decision variables can be categorized into this (non-exhaustive) list:

1. Power lines:
   - i Expansion of existing power lines (reconductoring)
   - ii The addition of new power lines
2. Generation:
   - i Expansion of existing power sources
   - ii The addition of new power sources
3. Flexibility:
   - i Addition of energy storage systems
   - ii Addition of demand-side response capacity

2-1-2 Objective cost function and strategy

In survey study [7], the cost functions and strategies grid expansion studies use are laid out. There are a lot of possibilities for the objective function, as well as from single-objective to multi-objective optimization schemes.

2-1-2-1 Cost function

One general theme in objective functions used is the overall minimization of costs:

$$\min c_{op} + c_{e,t} + c_{e,g} + c_{e,f} \tag{2-1}$$

Where c_{op} is the operational cost of the grid, and $c_{e,t}$, $c_{e,g}$, $c_{e,f}$ the expansion cost of transmission, generation and flexibility, respectively.

According to [6], a single-objective cost function is most common in grid expansion studies. It consists of a minimization of costs such as in equation 2-1. Expansion on this single-objective to a multi-objective cost function is most commonly done as an inclusion of a reliability index in the cost function. In figure 2-1, we see that around two-thirds of all studies use a single-objective cost function.

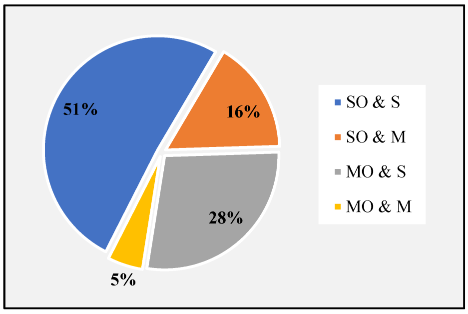

Figure 2-1: An overview of the fraction of studies using single- or multistage optimization and studies using single or multiobjective optimization strategies. [6]

2-1-2-2 Single- or multistage

In figure 2-1, we see that 79% of all studies employ a single-stage strategy, where modifications are all added simultaneously, over a multistage strategy where edges are added one-by-one.

2-1-3 Optimization method

There are a large nuber of possible approaches to the problem of grid expansion. Survey study [6] lays out the algorithms used to find the best candidate modification. There is a distinction between the algorithms used for power flow calculations and the algorithms used for the actual grid expansion, and we will introduce them in that order.

2-1-3-1 Optimal power flow

From survey study [7] we can also find that the largest part of grid expansion studies use a Monte-Carlo approach for power flow optimization. This often takes the form of computing an optimal power flow computation for each sample in a large set, and optimizing over the average of the resulting cost of that optimization. Examples of this, or similar techniques are found in papers [10, 11, 12, 13, 14, 15, 16, 17].

2-1-3-2 Grid expansion

The actual grid expansion optimization method varies between studies, with Branch and Bound / Branch and Cut methods being the largest category [6]. Some examples are:

1. Exact solution:
   i Branch and Bound / Branch and Cut [15, 16, 18, 19, 20, 17]
2. Approximate solution:
   i Greedy algorithm [21]
   ii Value-based algorithm [22]
   iii Neural network [23]
   iv Ant colony optimization [24]
   v Genetic algorithm [25]
   vi Artificial bee colony [26]

An overview of the distribution of all algorithms in grid expansion studies is shown in figure 2-2.

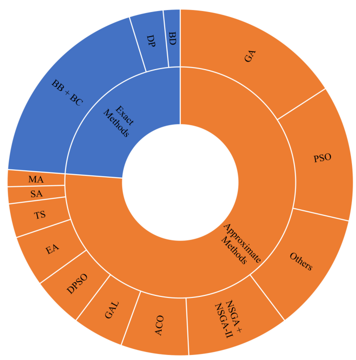

Figure 2-2: An overview of all optimization models used in existing studies. [6]

2-1-3-3 Data driven approaches in grid expansion

In grid expansion studies, data on the grid is used in making decisions on candidate modifications. In most studies, a random sample is either collected from historical data or generated using synthetic data generators trained on historical data [10, 12, 13, 15, 16, 27, 28]. Performance and/or reliability is then computed using these samples, and the most promising candidate modification is selected.

In other studies, more intricate information on the grid is known. This extra information can be used to determine the best modification, without the need for simulation. Examples of this approach are [29, 30, 31].

2-1-4 Reliability

In survey study [6] states that reliability is often described using an index in grid expansion studies, such as [12, 16, 31, 27]. This index is subsequently used in the optimization scheme.

- SAIDI ; Expected number of hours of interruption of an average customer
- SAIFI ; Sustained interruptions an average customer expects to occur
- ASAI ; Percentage of time an average customer is supplied without interruption
- AENS ; Energy not consumed due to interruptions
- ECOST ; Customer Interruption Cost (CIC) due to interruptions related to the distribution system

While these indices are a good indication of the reliability, they convey a less conclusive message on the robustness of the grid. Other papers utilize worst-case robustness as a means of improving reliability for their expansion planning optimization.

According to study [7], almost all papers compute reliability as a a-posteriori check, as it is not explicitly considered in the decision on modifications. Often, reliability is incorporated as a fraction of results that should be satisfying some constraints, but there is no real guarantee on or optimization aimed at improving this reliability.

A good example of this method can be found in paper [32], where the fraction of samples leading to excessive load curtailment is bounded and this condition is only checked after an expansion decision is made.

Papers [14, 20, 17], as an exception, explicitly incorporate risk into the multi-objective optimization problem of grid expansion. These papers consider low reliability as a financial risk factor, and include it as a financial cost into the optimization algorithm.

2-2 Robust control design

This section describes the process of scenario optimization. We start out with describing the concept of robust control, to then introduce the concept of level parameter ϵ, confidence parameter β and the associated requirement on a minimum number of samples N.

We introduce the concept of support constraints, their relevance to our research and provide the procedure of scenario optimization with discarding scenarios.

2-2-1 Robust control design

In this section, we introduce the concept of robust control design. We will shortly discuss worst-case robust control design, to continue with probabilistically robust control design. We then introduce the general form of the optimization problem. This section is mainly sourced from [8].

2-2-1-1 Worst-case Approach

In the field of control analysis and synthesis, it is well established to formulate the problems in terms of solutions to a convex optimization problem with linear matrix inequality constraints (LMI).

In a specific case, research is focused on situations where the data regarding the problem (for example the behavior of the plant) are uncertain. This research is then focused on finding a "guaranteed" approach, which satisfies the constraints for all admissible variations of this data. This is the notion of worst-case robust control.

In this case, one has to devise a solution that satisfies a possibly infinite number of constraints. A process which is not easily solvable and computationally intense. While there are a few methods of attacking this problem, such as introducing relaxations, the extend of which these methods influence the end-result are generally unknown and applying them in the first place requires a certain kind of dependence of the data on the underlying uncertainties.

2-2-1-2 Probabilistic Approach

Another method is the probabilistic approach. In this approach, we are no longer interested in satisfying all these constraints, all the time. We introduce a relaxation that the violation probability of this set of constraints is bounded by some variable $\epsilon \in (0, 1)$. In the case where we can make no assumptions on the underlying data, or attaining these conclusions using Monte-Carlo simulations is computationally intensive, we introduce some variable $\beta \in (0, 1)$ which describes the confidence in that our initial gauge on ϵ is faulty. In other words, it describes the probability that the ϵ-level we found for the samples taken this round is applicable to any sample in the sample-space. [33]

2-2-1-3 Problem formulation

The optimization problem we consider in general form is

$$\begin{aligned} &\min_{\theta \in \Theta} c^T \theta \\ &\text{subject to } \theta \in \Theta_i^\delta, i = 1, \ldots, N \end{aligned} \tag{2-2}$$

Where $\theta \in \Theta \subseteq \mathbb{R}^{n_\theta}$ is the "design parameter" of the problem, which includes all control variables and slack variables introduced in the problem. Because we are mostly interested in the feasibility of the system, a linear minimization using vector c is sufficient, other options will be discussed later. Then, for every optimization, we introduce uncertainty vector $\delta \in \Delta \subseteq \mathbb{R}^{n_\delta}$. This vector is the representation of the i.i.d. uncertainties with each iteration of the optimization, and shape the constraints.

The constraints are represented as sets $\Theta_i^\delta \subseteq \mathbb{R}^{n_\theta}, i = 1, \ldots, N$ for constraints 1 to N. Note that we can replace all these constraints by a single constraint $\Theta^\delta = \cap_{i=1,\ldots,N} \Theta_i^\delta$, since only one violated constraint is enough for our entire optimization to be unfeasible.

In the case of worst case design, the aim is to enforce all convex design constraints $\theta \in \Theta_i^\delta$ for all permissible values of $\delta \in \Delta$. However, due to the common occurrence that Δ has infinite cardinality, i.e. δ has an infinite amount of possible values, it is often computationally intensive or overly conservative to include all these possibilities into the pool of possible constraints on the optimization problem.

This is exactly why the concept of probabilistic design was introduced. Instead of finding θ that satisfies all constraints $\theta \in \Theta_i^\delta$ for all $\delta \in \Delta$, we include measure ϵ which acts as an upper bound on the probability of drawing a $\delta \in \Delta$ that results in one or more of the constraints being violated. This acts as a useful relaxation on the worst-case scenario as it allows for some leeway, making it far less computationally expensive and allows the designer to specify the conservatism of the approach to the optimization.

2-2-2 Scenario approach to robust control design

In this section, we build on the general form of the optimization problem previously introduced in equation 2-2, and formalize the measure of the violation probability and the scenario design algorithm. From there we formalize the definitions of level parameter ϵ and confidence level β. This section is sourced from [8].

2-2-2-1 Violation probability

From equation 2-2 we found that we can combine all constraints into a single constraint, without losing information on the feasibility of the problem:

$$\Theta^{\delta} = \cap_{i=1,\dots,N}\Theta_i^{\delta} \tag{2-3}$$

We now define the violation probability as the measure of the volume of parameters of $\delta \in \Delta$ that lead the problem to be infeasible. We define it as

$$V(\theta) \doteq \mathbb{P}\{\delta \in \Delta : \theta \notin \Theta^{\delta}\} \tag{2-4}$$

Where it is logical that a solution θ with a small non-zero associated $V(\theta)$ is feasible for most samples $\delta \in \Delta$. Therefore, this solution is *approximately feasible* for the robust optimization problem.

2-2-2-2 Scenario design

Assume now, that we want to check the design not for a single sample vector $\delta \in \Delta$, but for N independent identically distributed (i.i.d.) sample vectors $\delta^{(1)}, \dots, \delta^{(N)}$ drawn according to probability Prob. We define the convex optimization problem as

$$\begin{aligned} \mathrm{RCP}_N : &\min_{\theta\in\Theta} c^T\theta \\ &\text{subject to } \theta \in \Theta^{(\delta_j)}, j = 1, \dots, N \end{aligned} \tag{2-5}$$

Where $\Theta^{(\delta_j)}$ is defined similarly as in equation 2-2 and 2-3, namely
$\Theta^{(\delta_j)} = \cap_{i=1,\dots,N}\Theta_i^{(\delta_j)}$.

We can conclude that by combining some N drawings of the sample vectors, and checking for feasibility on all these samples, we end up with an easily computable optimal design parameter, only having to deal with finite number N constraints, alleviating the concerns with the worst-case approach.

2-2-2-3 Level parameter and confidence level

Effectively assessing the violation probability of the underlying design is now directly linked to the specific samples we draw in the Scenario design. We therefore opt to bound the violation probability by a certain value, instead of making claims on the exact value.

We define the level parameter $\epsilon \in (0,1)$. We say that $\theta \in \Theta$ is a ϵ-level solution if $V(\theta) \leq \epsilon$. This level parameter effectively acts as an upper bound to the violation probability.

Because this violation probability is still bound to the underlying samples drawn, we introduce the confidence parameter β which describes the probability that the ϵ-level we found for the samples taken this round is applicable to any sample in the sample-space.

$$\mathbb{P}^N\{V(\theta) \leq \epsilon\} \geq 1-\beta \tag{2-6}$$

The next section will delve deeper into the relationship between the amount of samples N needed and the level- and confidence parameter chosen for the experiment.

2-2-3 Sample Sizes

In this section we explain the relationship between the number of samples drawn N, and the specified level- and confidence parameters, ϵ and β respectively. We finish the section with a more concrete interpretation of the scenario approach to robust control design. This section is sourced from [8].

At this point in time, a lot of research is focused on lowering the number of required samples needed to infer a conclusion on ϵ and β. A very simple bound is defined linearly:

$$N \geq N_{lin}(\epsilon, \beta, n_\theta) \doteq \left\lceil \frac{n_\theta}{\epsilon\beta} - 1 \right\rceil \tag{2-7}$$

Where our samples drawn must be at least equal to this bound. However, this bound is linear in both ϵ^{-1} and β^{-1}, and since typically β is chosen very small, this bound is less then ideal. This bound has been improved to be only logarithmically dependent on β:

$$N \geq N_{gen}(\epsilon, \beta, n_\theta) \doteq \left\lceil \inf_{\nu \in (0,1)} \frac{1}{1-\nu} \Big(\frac{1}{\epsilon} \ln \frac{1}{\beta} + n_\theta + \frac{n_\theta}{\epsilon} \ln \frac{1}{\nu\epsilon} + \frac{1}{\epsilon} \ln \frac{(\frac{n_\theta}{e})^{n_\theta}}{n_\theta !} \Big) \right\rceil \tag{2-8}$$

Where we naturally take the value of ν in the range $(0,1)$ to have our bound be the lowest. This can be simplified into a more direct relationship, with the concession of our new bound being at most a factor 2 larger than N_{gen} from equation 2-8:

$$N \geq N_{log}(\epsilon, \beta, n_\theta) \doteq \left\lceil \frac{2}{\epsilon} \ln \frac{1}{\beta} + 2n_\theta + \frac{2n_\theta}{\epsilon} \ln \frac{2}{\epsilon} \right\rceil \tag{2-9}$$

This new bound N_{log} is orders of magnitude lower than the bound N_{lin} from equation 2-7. Typically, ϵ is chosen very small, say 0.1, and n_θ is chosen to be 10. We can see in figure 2-3 that the sample sizes for N_{lin} are a lot larger than for N_{log}, the penalty paid for having a easier to calculate sample size from N_{gen} is comparatively small.

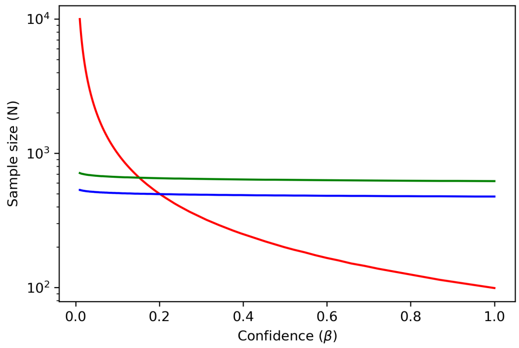

Figure 2-3: Comparison between N_{lin} (red), N_{gen} (blue) and N_{log} (green) for various values of β; $\epsilon = 0.1$, $n_\theta = 10$

To conclude, if we have an optimization problem that is feasible for at least the N samples drawn, we can state with confidence β that the solution is at least ϵ-level robust (i.e. will fail for at most ϵ of samples with confidence $1 - \beta$).

2-2-4 Support constraints

In this section we touch on the concept of support constraints, and with this the measure of complexity. We will use these concepts later on in order to introduce a way of calculating the value for ϵ with these. This section is sourced from [34, 35, 36, 9].

2-2-4-1 Support constraints

A support constraint is a constraint from a scenario $\Theta^{(\delta_j)}$ with index k that, if that scenario were to be removed, the optimal solution would improve (lower cost value). In terms of the problem formulation:

$$\begin{aligned} \hat{\theta}_N &= \arg\min_{\theta \in \Theta} c^T \theta \\ &\text{subject to } \theta \in \Theta^{(\delta_j)}, \forall j \in \{1, \dots, N\} \\[1em] \hat{\theta}_{N \setminus K} &= \arg\min_{\theta \in \Theta} c^T \theta \\ &\text{subject to } \theta \in \Theta^{(\delta_j)}, \forall j \in \{1, \dots, N\} \setminus K \\[1em] c^T \hat{\theta}_{N \setminus K} &< c^T \hat{\theta}_N \end{aligned} \tag{2-10}$$

Which is analogous to equation 2-2. It is shown that the number of support constraints $|K|$ is always less or equal to the dimension of the optimization variable, where the former is annotated by k, also known as the complexity of the problem, and the latter by n_θ. Whenever the number of support constraints is equal to the dimension of the optimization variable, the problem is considered *fully-supported*.

We call a problem *Non-degenerate* if the solution with all constraints and the solution with only the support constraints coincide with probability 1 (with sample set $\{\delta^{(1)}, \delta^{(2)}, \ldots, \delta^{(N)}\}$). This is assumed from now on.

2-2-4-2 Level parameter from support constraints

It is possible to compute an a-posteriori level parameter from the number of support constraints found. This is done by fixing our confidence parameter β, and running for N scenarios. We then find k support constraints, and we can find the level parameter as

$$0 = \frac{\beta}{N+1} \sum_{m=k}^{N} \binom{N}{k} (1-\epsilon_k)^{m-k} - \binom{N}{k} (1-\epsilon_k)^{N-k} \tag{2-11}$$

Which is a polynomial with one solution for $\epsilon_k \in (0,1)$. This is the revised level parameter, based on the number of value support constraints. This new way of computing the level parameter especially improves the result for a low number of support constraints.

This new function allows the level parameter to be calculated a posteriori, calculating the level parameter from results of the optimization, where the previous functions defined ϵ and β a priori, resulting in a minimal number of samples. How we can exploit this, we will explain in the next section.

2-2-5 Distribution of violation probability

Equation 2-11 follows from a beta distribution in the number of samples and number of support constraints. The claim that this holds, is made in [34], and we show that we comply with the necessary assumption in appendix 6-1-1. This beta distribution has the following probability density function:

$$\begin{aligned}
\mathbb{P}^N\{V(\theta) \leq \epsilon\} &= 1 - \sum_{i=0}^{k-1} \binom{N}{i} \epsilon^i (1-\epsilon)^{N-i} \\
f_{V(\theta)}(\epsilon, k)\Big|_N &= \sum_{i=0}^{k-1} \binom{N}{i} \epsilon^{i-1} (1-\epsilon)^{N-i-1} (\epsilon(N-2i)+i) \\
&= C \cdot \epsilon^{k-1}(1-\epsilon)^{N-k}
\end{aligned} \tag{2-12}$$

With C a normalization constant such that $\int_0^1 f_{V(\theta)}(\epsilon, k) d\epsilon = 1$. It is this relationship that allows us to more accurately compare two different violation probabilities on their density functions, given the respective complexity of each optimization problem.

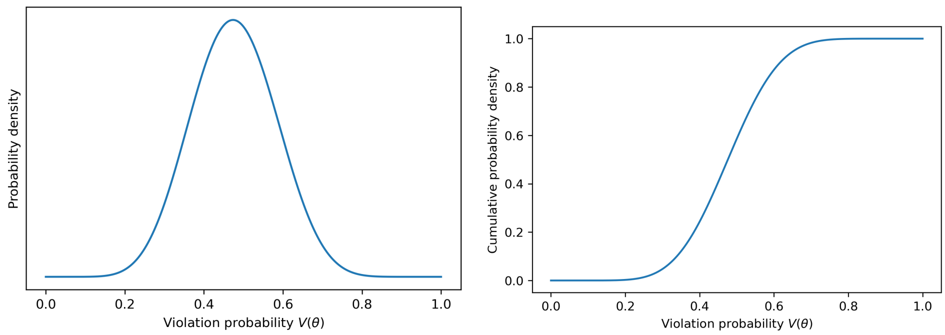

Figure 2-4: Probability density function and cumulative probability function for $N = 20$ and $k = 10$

2-2-6 Discarding scenarios

In this section we introduce the final step in accessing more performance using the same level parameter: discarding constraints. We introduce the workings of discarding constraints, and this results in a final expression for the level parameter. This section is sourced from [9].

2-2-6-1 Discarding algorithm

Suppose the optimization is run, but there are some R scenarios which heavily limit the feasibility of the problem (i.e. result in a disproportionate limiting of the solution space). We can then introduce algorithm $\mathcal{A}(\cdot)$ such that applying that algorithm to our scenarios, it would find the same amount of R constraints. We can then trade off performance, for a higher a-posteriori level parameter (which is still below the originally set a-priori value).

Any algorithm could satisfy this, but the only requirement is that the constraints selected are almost surely violated. We can check for this by checking if the solution found with these constraints discarded would violate those same constraints. If not, we can remove other constraints.

2-2-6-2 Level parameter from support constraints and discarding algorithm

We can combine this algorithm with the information on support constraints in equation 2-10, resulting in:

$$0 = \frac{\beta}{N+1} \sum_{m=k}^{N} \binom{N}{k} (1-\epsilon_k)^{m-k} - \binom{N+R}{R} \binom{N}{k} (1-\epsilon_k)^{N-k} \tag{2-13}$$

Which is again a polynomial with one solution for $\epsilon_{k,R} \in (0,1)$. This is the revised level parameter, based on the number of found support constraints and discarded scenarios. Note that, as a consequence of $\binom{N+R}{R}$, our newly found level parameter is higher than the one found in equation 2-10. This then allows us to more accurately find the solution that fits our desired risk level, by tuning the number of discarded scenarios.

2-2-7 Robust grid operation

In this section, we finally provide the full optimization scheme to accurately find the optimal performance and gauge the feasibility of an energy grid. The exact workings of the energy grid will be discussed in chapter 2-3.

2-2-7-1 Optimization loop

We start out with a grid graph setup $\mathcal{G}$, from which we can find the values for V_0, Z_p, and Z_q. We then choose a suitable optimization function and suitable constraints. This optimization is similar to [8]. We apply the following steps to optimize:

1. Pick values for $\epsilon \in (0, 1)$ and $\beta \in (0, 1)$ for the level and confidence parameter.
2. Use equation 2-9 to find N_{log} the lower limit on samples needed to conclude ϵ and β.
3. Pick $N \geq N_{log}$ as the number of samples.
4. Generate $\{\delta^1, \delta^2, \cdots, \delta^N\} \in \Delta^N$ as i.i.d samples for our active and reactive sampled loads scenarios.
5. Solve the optimization scheme according to the chosen objective function and constraints.
6. This is either feasible, or not feasible.
   i If it is feasible, we conclude the optimization is ϵ-level feasible with confidence β.
   ii If it is not feasible, no conclusion can be made.

2-2-7-2 Discarding constraints

We can improve our control, i.e. move closer to the allowed violation probability, by discarding constraints. A prerequisite of discarding these constraints is that these discarded scenarios are almost surely violated.

To find these, we find the scenarios that will deviate voltages the most from the nominal value. Exactly how to calculate these voltage levels $|v_{\mathcal{L}}|$ will be discussed in the next chapter. It are these scenarios that will violate a voltage constraint the first. In case of a voltage constraint such as in the first option at 2-26, we can find them using equation 2-14. For other constraints, the method would look similar.

$$M = \{\underset{i \in \{1,\ldots,N\}}{\arg\max} \, |v_{\mathcal{L}}|_l^i, l \in \{1, \ldots, n\}\} \cup \{\underset{i \in \{1,\ldots,N\}}{\arg\min} \, |v_{\mathcal{L}}|_l^i, l \in \{1, \ldots, n\}\} \tag{2-14}$$

Where $|v_{\mathcal{L}}|_l^i$ is the voltage level at bus l with sample i as a result of grid operation. M is the set of scenarios where the voltage on any of the busses is either maximal or minimal. We then check if each of these scenarios is actually a support constraint using equation 2-10. If they are, they are kept as a support constraint in the set

$$I = \{m \in M : c^T \theta^*_{N,m} < c^T \theta^*_N\} \tag{2-15}$$

We can now optimize, using the following optimization loop, which is also the one used in [9].

1. Pick values for $\epsilon \in (0,1)$ and $\beta \in (0,1)$ for the level and confidence parameter.
2. Use equation 2-9 to find N_{log} the lower limit on samples needed to conclude ϵ and β.
3. Pick $N \geq N_{log}$ as the number of samples.
4. Generate $\{\delta^1, \delta^2, \cdots, \delta^N\} \in \Delta^N$ as i.i.d samples for our active and reactive sampled loads scenarios.
5. **Set the set of discarded scenarios as empty,** $I_{discarded} = \emptyset$
6. Solve the optimization scheme according to the chosen objective function and constraints.
7. This is either feasible, or not feasible.

   i If it is feasible, we can conclude that the optimization is ϵ-level feasible with confidence β. Continue.

   ii If it is not feasible, no conclusion can be made. Stop.
8. **Observe the number of support constraints using set** I **from equations 2-14 and 2-15**
9. **Calculate the number of discarded scenarios** R **by using equation 2-13 and** $\epsilon(k,R) \leq \epsilon < \epsilon(k,R+1)$
10. **Use the set of support constraints** I **to pick** R **scenarios to discard and add these to the set of discarded scenarios.** $I_{discarded} = I_{discarded} \cup \{i_1, \ldots, i_R\}$
11. **Now, we either improved our solution or it remained the same**

    i **If the number of support constraints no longer changes, and all removed constraints are violated. Continue**

    ii **Otherwise, add all constraints back from** $I_{discarded}$ **that were not violated and return to optimization step 6 using** $\{\delta^1, \delta^2, \cdots, \delta^N\} \backslash I_{discarded}$.
12. **If the** $I_{discarded}$ **is unchanged from the previous iteration, we are done.**

2-3 Grid operation

In this section, we showcase the chosen power flow model and how to describe the model using the connections on the grid.

We start with basic concepts of active and reactive power flow and grid operation in general, and continue with our chosen model and the options for constraints on the operation of the grid.

2-3-1 Basic principles of grid operation

In this section we take a look at the power grid. We provide a basic description of the power grid, and introduce some limitations, on which we will elaborate later. The information found in this section is found at [37].

2-3-1-1 Power on the grid

A lot of products that we use consume electric power. This power has to be provided to consumers from the grid. In order to provide electricity to consumers, grid operators try to continually balance the power inputted and outputted into the grid. If too little power is provided, electric appliances may operate worse or a blackout will ensue. Too much power on the grid will also lead to issues with regards to the grid.

The grid operator balances the power in with power out by monitoring the voltage and the frequency of power on the circuit. The European grid runs on 50 Hz, with 220 Volts coming out the outlets. Whilst the voltage can be adjusted throughout the grid, which is utilized because transporting electricity at high voltage means a lower current is necessary for the same power, the frequency over the grid is kept constant.

In order to improve the capability of grid operators to keep the grid balanced, both predicting algorithms as well as dispatch-able balancing volumes and market forces are used.

2-3-1-2 Power through the grid

In order to get from the producer to the consumer, the grid operator distributes the power through the power lines of which the grid consists. These cables have certain limits on the voltage and power they can transmit, so it is up to the grid operator to install enough capacity such that these physical limits are as small a problem possible. This is called *decongestion.*

2-3-1-3 Transmission and distribution networks

There is a distinction between transmission system operators (TSO) and distribution system operator (DSO). The TSO is responsible for the high-voltage, high-power transmission of electricity, mostly to the DSO who transforms it into lower voltage electricity to eventually be delivered to the consumer.

Both these grid operators are responsible for balancing their own electricity grid, as well as alleviating congestion.

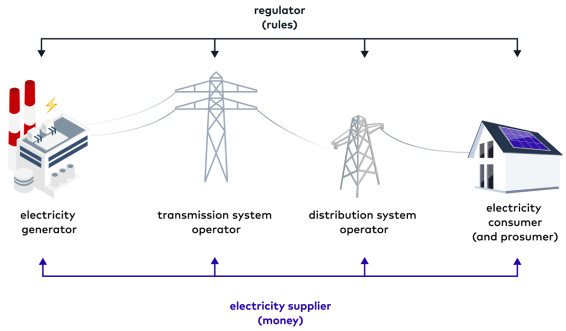

Figure 2-5: Power is (conventionally) added to the high-voltage transmission network, subsequently transported to the distribution network, to then arrive at the consumer. [37]

2-3-1-4 Reactive power

When an alternating voltage is applied to a circuit, the current is not necessarily in phase with the current. When there are reactive components such as inductors and capacitors, there is a phase shift. This phase shift leads to the power draw not consistently being from source to drain (strictly positive power).

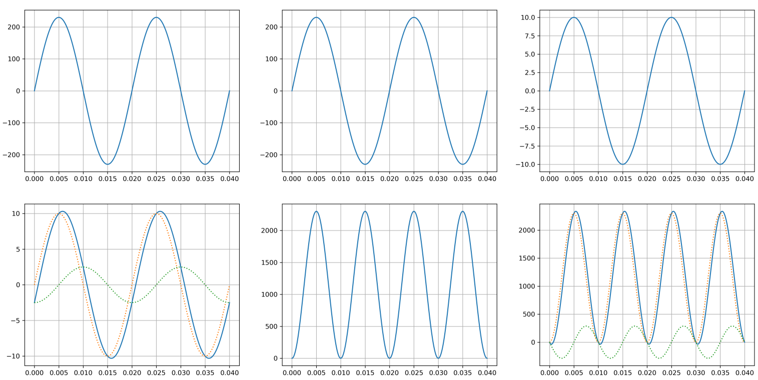

Figure 2-6: Two examples of power flow for a fully active (top) and partially reactive (bottom) power flow. In the leftmost figures, the voltage is denoted, the middle corresponds to the current, and the rightmost figures corresponds to the power flow. The orange and green dashed lines correspond to active- and reactive components, respectively.

We see in row three of figure 2-6 that there is a certain component of the power that is not delivered to the drain. In order to still calculate the actual power delivered, we separate the current into two components. The active current, being in phase with the voltage, and the reactive current, being out of phase by 90 degrees ($\frac{\pi}{2}$). Together these two components add up to the actual current, and they are perpendicular:

$$\langle \sin(x+0), \sin(x+\frac{\pi}{2})\rangle = \int_0^\pi \sin(x+0)\cdot\sin(x+\frac{\pi}{2})dx = 0 \quad (2\text{-}16)$$

By plotting the power components related to these current components, as in we see in figure 2-6, that the active power P is now again strictly positive, and corresponds to a continuous flow of energy from source to drain, and the reactive power Q results in no net flow, only oscillating between the source and drain.

Reactive power is useful since some components actually require the current to be leading the voltage a little, but too much reactive power results in unnecessary loss due to a lot of power being dissipated by oscillating over imperfect and inefficient elements.

2-3-2 Coupled power flow model

In this section, we formulate the grid into a directed graph, with the power and voltage laws. This mathematical description will then later be used as guidance on how to gauge - and improve upon - the performance of a grid. The derivation will follow the same steps as the *Linear coupled power flow model* from [38] and will result in the same equation as in [9].

2-3-2-1 Graphs

We can construct power grids as a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{W})$. In this graph, the set of nodes $\mathcal{V} = \{1, \dots, N_{bus}\}$ contain all points where power is produced, consumed or distributed. This distribution is done through the edges in the set $\mathcal{E} = \{(v_i, v_j)|v_i, v_j \in \mathcal{V}\}$, corresponding to the cables between (some of) the nodes. In addition, each edge has a weight (admittance) defined by its element in the set $\mathcal{W} = \{w_{i,j}|(v_i, v_j) \in \mathcal{E}, w_{i,j} \in \mathbb{C}\}$.

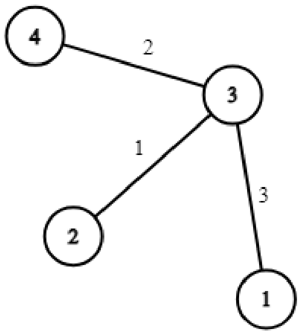

Figure 2-7: An example of a graph with graph weights displayed next to the edges.

2-3-2-2 Power dynamics on edges and nodes

Suppose we create a grid $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{W})$ with n nodes. For each edge, or power cable, we can then construct Ohms law over that edge:

$$i = Yv \tag{2-17}$$

Where $i \in \mathbb{C}^n$ and $v \in \mathbb{C}^n$ are the vectors of injected current and voltage at each node. If we assume the shunt-admittances at the busses are negligible. Y coincides with the weighted Laplacian of the graph describing the grid, with edge weights equal to the admittance of the power lines. We construct it as follows:

$$\begin{aligned} & Y \in \mathbb{C}^{n\times n} \\ & Y_{i,j} = \begin{cases} w_i + w_{sh,i} & \text{if } i = j \\ -w_{i,j} & \text{if } (v_i, v_j) \in \mathcal{E} \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{2-18}$$

Where $w_i = \sum_{\{j \in \mathcal{V} | (v_i, v_j) \in \mathcal{E}\}} w_{i,j}$ or the sum of the admittance values of all incoming and outgoing cables. Shunt admittances $w_{sh,i}$ will be considered negligible. Applying this algorithm to the graph shown in figure 2-7, we find.

$$Y = \begin{bmatrix} 3 & 0 & -3 & 0 \\ 0 & 1 & -1 & 0 \\ -3 & -1 & 6 & -2 \\ 0 & 0 & -2 & 2 \end{bmatrix} \tag{2-19}$$

We then add a slack node anywhere in our grid and connect it to another node. This new node is allotted index 0 and the grid including matrix Y is updated accordingly. We use the set $\mathcal{L}$ to denote the grid excluding this slack node.

This slack node is used to impose some steady-state voltage, defined by

$$v_0 = V_0 e^{j\phi_0} \tag{2-20}$$

With known amplitude $V_0 \geq 0$ and phase $-\pi < \phi_0 \leq \pi$. We can then split our matrix Y into the slack node and the rest of the grid:

$$\begin{bmatrix} i_0 \\ i_{\mathcal{L}} \end{bmatrix} = \begin{bmatrix} Y_{00} & Y_{0\mathcal{L}} \\ Y_{\mathcal{L}0} & Y_{\mathcal{L}\mathcal{L}} \end{bmatrix} \begin{bmatrix} v_0 \\ v_{\mathcal{L}} \end{bmatrix} \tag{2-21}$$

Here, assuming that the grid is a connected graph, $Y_{\mathcal{L}\mathcal{L}}$ is invertible. We obtain

$$v_{\mathcal{L}} = v_0 \mathbb{1} + Y_{\mathcal{L}\mathcal{L}}^{-1} i_{\mathcal{L}} \tag{2-22}$$

Where we see that, indeed, adding the slack bus resulted in a baseline voltage level, but otherwise does not deviate the dynamics from equation 2-17. We model all nodes in $\mathcal{L}$ as PQ buses, and therefore we can define the imposed complex power vector $s_{\mathcal{L}}$ as

$$s_{\mathcal{L}} = \text{diag}(\overline{i_{\mathcal{L}}})v_{\mathcal{L}} \tag{2-23}$$

Here, $(\bar{\cdot})$ of a vector is the vector consisting of complex conjugate pairs of the original vector. Using this, we can find the approximations of the magnitudes of voltage on each node in the grid, under the assumption that the voltage deviations are much smaller than the nominal voltage V_0.

$$|v_{\mathcal{L}}| = V_0\mathbb{1} + \frac{1}{V_0}\text{Re}(Y_{\mathcal{L}\mathcal{L}}^{-1}\overline{s_{\mathcal{L}}}) \tag{2-24}$$

We can then redefine $Y_{\mathcal{L}\mathcal{L}}^{-1}$ as impedance matrix Z and split that matrix into Z_P and Z_q for active- and reactive power, respectively.

$$|v_{\mathcal{L}}| = V_0\mathbb{1} + \frac{1}{V_0}(Z_p(P_\delta + P_\theta) + Z_q(Q_\delta + Q_\theta)) \tag{2-25}$$

Where P_δ and Q_δ represent the active- and reactive sampled loads (i.e. the samples for our scenario optimization), and P_θ and Q_θ represent the active- and reactive controlled loads (i.e. the control variables), respectively.

2-3-3 Objective functions

In our optimization process, we can choose from a host of objective functions, depending on our eventual goal.

- If we only care about violation probability, a simple linear vector multiplication should suffice. ($\min c^T\theta$). This cost function is commonplace. [9, 39]
- If we wanted to maximize the share of green electricity use, we would apply weights to the multiplication. ($\min c_{green}^T\theta$)
- If we wanted to provide the cheapest power, a differently weighted vector may be applicable. ($\min c_{cost}^T\theta$)
- Any combination is possible, but increasing complexity of the cost function will have effects on the computation time.

2-3-4 Constraints

We will now look at possible options for constraining this grid.

Note that these are not formalized in the standard form of $\theta \in \Theta_j^\delta$. This is done to improve legibility, but they can easily be reconstructed as such. These constraints are selected from [7, 40].

2-3-4-1 Constraints on the voltage levels

This vector $|v_{\mathcal{L}}|$ from before can be used for constraints on the power grid, either in the form of a hard limit on the voltage level at each node, a limit on the voltage delta between two nodes, or apparent power flow based on the voltage delta calculated at each edge. These possible options are written as

$$\begin{cases} V_{min} \leq |v_{\mathcal{L}}| \leq V_{max} & \text{Limit on voltage level} \\ ||v_{\mathcal{L}}|_i - |v_{\mathcal{L}}|_j| \leq \Delta V_{max} & \text{Limit on voltage spread} \\ ||v_{\mathcal{L}}|_i \overline{(|v_{\mathcal{L}}|_i - |v_{\mathcal{L}}|_j)\overline{w_{i,j}}}| \leq S_{i,j}^{max} \; \forall w_{i,j} \in \mathcal{W} & \text{Limit on apparent power flow} \end{cases} \tag{2-26}$$

2-3-4-2 Constraints on the power levels

In some cases, the total power load for a node is also limited, which simply limit the sums of active- and reactive powers. These constraints would look like

$$\begin{cases} P_{min} \leq P_{\theta} \leq P_{max} & \text{Limit on active power} \\ Q_{min} \leq Q_{\theta} \leq Q_{max} & \text{Limit on reactive power} \end{cases} \tag{2-27}$$

2-3-4-3 Ramping constraints

Another constraint, used less often in optimal power flow studies and only relevant when considering power flow over time, are ramping constraints. These limit the fluctuation in power draw and generation for nodes that are (physically) unable to fluctuate faster. It can be formalized as follows, and only limits active power loads.

$$|\Delta P_{\delta} + \Delta P_{\theta}| \leq R^{max} \text{ Where } \Delta P_{(\cdot)} = P_{(\cdot)}^{t} - P_{(\cdot)}^{t-1} \tag{2-28}$$

2-3-4-4 Curtailment

Some electricity sources, that have been sampled thus far, can actually be curtailed. This means that the power plant output can be somewhat steered, where the bandwidths given in equation 2-27 are determined by the sampled output we used before. This effectively means we increase the control dimension.

2-3-4-5 Demand-side response

Demand-side response (DSR) is a measure made by a grid operator to free up capacity. For each load, there is three options in terms of DSR capability:

1. There is no DSR possible.
2. There is DSR possible, but only for this time period.
3. There is DSR possible, and load needs to be shifted to another time period.

Chapter 3

Experimental design

3-1 Optimization model

In this section, we describe the entire design process and considerations of this thesis.

We start with the considerations on the model, given our research aim and choose a model architecture. Second, we describe the two options of leveraging the number of support constraints in optimizing the grid, and motivate our eventaul choice. Third, we introduce the development horizon and associated parameters depth D and breadth B. Given the model chosen and the design aspects presented, we describe the cost functions and optimization loops used in this thesis.

3-1-1 Optimization model considerations

As denoted in figure 2-1, there are a combination of single- or multi-stage and single- or multi-objective optimization function. In this section, we comment on the strategies chosen and motivate our choices. We combine this strategy with an algorithm from figure 2-2 and motivate our choice.

Ultimately we opt to implement a multi-stage, multi-objective optimization strategy, implemented using a greedy approach. By extending the development horizon, we move closer towards a full Branch-and-Bound approach.

3-1-1-1 Model philosophy

In this study, we want to compare performance between optimization methods. For this comparison, the algorithm has to contain some specific attributes:

- Have a large set of candidate modifications as to promote exploration and differentiation between simulation results.
- Let the optimization play out as to find which algorithm is more likely to converge to a better local optimum.
- Include reliability as an explicit influence on optimization.
- Be able to run a simulation in a reasonable time frame.

These criteria ultimately confine us to a small subset of all possible optimization strategies and models.

3-1-1-2 Optimization model

Given the model philosophy and considerations of the previous section, we design our grid expansion optimization with the architecture described in this section.

Multi-stage optimization

Since we want a large search space for the grid expansion program, we want to find the effect of adding multiple modifications to the grid, and lastly we want to compute this in feasible time, we opt for a multi-stage approach. By adding modifications one at a time, and not consider all combinations of modifications, we limit the number of simulations needed per modification step as much as possible whilst still allowing for exploration.

Multi-objective optimization

We follow the general consensus from [6] to formulate the optimization function as a cost minimization similar in spirit to equation 2-1. Since we want to include reliability as an explicit variable in optimization, we add a specific term that uses a quantity pertaining to the reliability of the grid in the cost function. This results in us employing a multi-objective optimization function.

Greedy optimization model

As we are checking a large set of candidate modifications, we want to keep the number of simulations at an acceptable level. This is why, for the base case, we use a greedy, multi-stage approach. This model choice allows for an acceptable number of simulations, whilst still being able to let optimization runs run their course.

Decision variables

To keep the number of possible modifications bounded, we opt for a grid expansion program that only considers upgrading existing grid connections or adding new power lines as decision variables. This is also in line with the currently most pressing issue for distribution and transmission grid operators. [41]

3-1-2 Gauging reliability of grid using support constraints

In this section, we describe two methods of exploiting the complexity of the grid operation optimization, and motivate our choice.

3-1-2-1 Comparing the violation probability of two a-posteriori results

Where the operation of electricity grid is mainly concerned with keeping the violation probability below some bound, we actually have more information about the distribution of that violation probability. We can exploit this information to extract information about improvements in terms of violation probability.

Because the violation probability of both results exist on a probability density function as given in equation 2-12, we can describe the probability of a random sample from the first being higher than a random sample from the other, with some margin γ.

$$\mathbb{P}^N\{V_2(\theta_2) - V_1(\theta_1) \leq -\gamma\} = \int_0^1 f_{V_1(\theta_1)}(V_1, k_1) \int_0^{V_1-\gamma} f_{V_2(\theta_2)|V_1(\theta_1)}(V_2, k_2) dV_2 dV_1 \tag{3-1}$$

Where $V_{(\cdot)}(\theta)$ are the two violation probabilities and the left part of the equality describes the probability that violation probability $V_2(\theta)$ is smaller than $V_1(\theta)$ by margin γ. Since the effect of this margin differs by a lot when comparing from very small violation probabilities to large ones, we opt to set γ to 0. f_{V_1} and $f_{V_2|V_1}$ describe the a-posteriori (conditional) probability density functions of the violation probabilities $V_1(\theta)$ and $V_2(\theta)$, give the N samples used.

If we use two graphs that are fairly similar to determine $f_{V_1(\theta_1)}$ and $f_{V_2(\theta_2)|V_1(\theta_1)}$, the logical assumption would be that the violation probability $V_2(\theta_2)$ is not independent of $V_1(\theta_1)$. However, the exact probability density function $f_{V_2(\theta_2)|V_1(\theta_1)}$ is hard to compute. For the time being, we will assume independence, and test if this assumption improves results.

We can now employ the expression for $f_{V(\theta)}$ from equation 2-12 and write equation 3-1 as

$$\begin{aligned}\mathbb{P}^N\{V_2(\theta_2) \leq V_1(\theta_1)\} &= \frac{1}{C_1 C_2} \int_0^1 x^{k_1-1}(1-x)^{N-k_1} \int_0^x y^{k_2-1}(1-y)^{N-k_2} dy dx \\ &= h_{pdf}(N, k_1, k_2)\end{aligned} \tag{3-2}$$

Where k_1 and k_2 are the complexities of problem 1 and 2, respectively. C_1 and C_2 are normalization constants as in equation 2-12. N denotes the number of samples. Since the probability density functions of $V_1(\theta_1)$ and $V_2(\theta_2)$ have continuous domains, the probabilities $\mathbb{P}^N\{V_2(\theta_2) \leq V_1(\theta_1)\}$ and $\mathbb{P}^N\{V_1(\theta_1) \leq V_2(\theta_2)\}$ are complement and thus $h_{pdf}(N, k_1, k_2) = 1 - h_{pdf}(N, k_2, k_1)$.

Optimizing over probability density curves of grid reliability
Equation 3-2 allows us to provide an expression on the improvement in robustness, purely based on the number of samples and the a-posteriori complexities of both optimizations. This allows us to compare the reliability of two graphs using the proxy of simulation results.

$\mathbb{P}^N\{V_{\mathcal{G}+u} \leq V_{\mathcal{G}}\}$ describes the probability that the violation probability of the original graph $\mathcal{G}$ is greater or equal than the violation probability of the graph modified by u. This probability can be computed with equation 3-2, using the number of support samples of optimization problems using both grids.

This computation is quite expensive to run numerically, but since we know the number of samples and the size d of the decision variable θ, we can compute values for $h(N, k_1, k_2)$ for all possible combinations of complexities $k_{\mathcal{G}}$ (k_1) and $k_{\mathcal{G}+u}$ (k_2) upto n_θ and store it for later:

$$H_{pdf} = \begin{bmatrix} h_{pdf}(N,0,0) & 1-h_{pdf}(N,0,1) & \cdots & 1-h_{pdf}(N,0,1) \\ h_{pdf}(N,0,1) & h_{pdf}(N,1,1) & \cdots & 1-h_{pdf}(N,1,n_\theta) \\ \vdots & \vdots & \ddots & \vdots \\ h_{pdf}(N,0,n_\theta) & h_{pdf}(N,1,n_\theta) & \cdots & h_{pdf}(N,n_\theta,n_\theta) \end{bmatrix} \tag{3-3}$$

3-1-2-2 Comparing a-posteriori epsilon level

Another option of using the number of support constraints as a measure of robustness, is provided in equation 2-11. In this equation, an a-posteriori epsilon is defined using β, N and the complexity of the problem. We define

$$\begin{aligned} \epsilon_{k_1} \in (0,1) : 0 &= \frac{\beta}{N+1}\sum_{m=k_1}^{N}\binom{N}{k_1}(1-\epsilon_{k_1})^{m-k_1} - \binom{N}{k_1}(1-\epsilon_{k_1})^{N-k_1} \\ \epsilon_{k_2} \in (0,1) : 0 &= \frac{\beta}{N+1}\sum_{m=k_2}^{N}\binom{N}{k_2}(1-\epsilon_{k_2})^{m-k_2} - \binom{N}{k_2}(1-\epsilon_{k_2})^{N-k_2} \\ h_{post}(N,k_1,k_2) &= \left.\frac{\epsilon_{k_2}-\epsilon_{k_1}}{\epsilon_{k_1}}\right|_{N,\beta,k_1,k_2} \end{aligned} \tag{3-4}$$

This function $h_{post}(N, k_1, k_2)$ allows us to compare two results on a-posteriori level parameters, calculated using complexities. Similar to 3-3, we compute values for all possible combinations of complexities $k_{\mathcal{G}}$ (k_1) and $k_{\mathcal{G}+u}$ (k_2) up to n_θ and store it:

$$H_{post} = \begin{bmatrix} h_{post}(N,0,0) & h_{post}(N,1,0) & \cdots & h_{post}(N,n_\theta,0) \\ h_{post}(N,0,1) & h_{post}(N,1,1) & \cdots & h_{post}(N,n_\theta,1) \\ \vdots & \vdots & \ddots & \vdots \\ h_{post}(N,0,n_\theta) & h_{post}(N,1,n_\theta) & \cdots & h_{post}(N,n_\theta,n_\theta) \end{bmatrix} \tag{3-5}$$

3-1-2-3 Implemented measure of reliability improvement

We have formulated two different methods of gauging an improvement in reliability, both using the number of support constraints. In this section, we motivate our choice for the former.

Comparing the two methods can be done by comparing the relative values of H_{pdf} and H_{post}. In figure 3-1, we can see these values normalized such that $h_{pdf} \in [-1, 1]$ and $h_{post} \in [-1, 1]$ $\forall k_1, k_2 \in \{0, \ldots, n_\theta\}$.

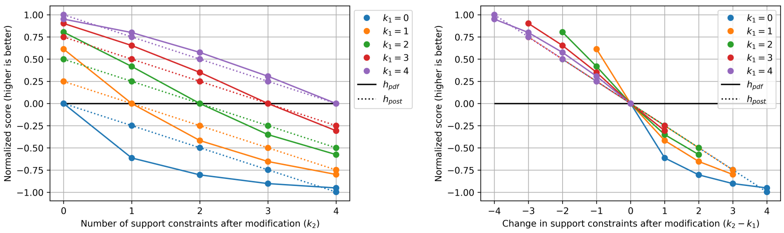

Figure 3-1: Normalized values for H_{pdf} and H_{post} for different combinations of k_1 and k_2, $N = 1209, n_\theta = 4$. The left-hand figure shows values as a function of the complexity of the graph after modification k_2, whereas the right-hand picture describes shows values as a function of change in complexity after modification $k_2 - k_1$.

In figure 3-1 we see that using the probability density function has two distinct advantages:

- The values of H_{pdf} encourage more complexity improvements around the level of complexity of the current grid.
- A complexity improvement closer to $k_1 = 0$ is rewarded more than an improvement closer to fully supported $k_1 = n_\theta$.

It is because of this reasons we opt for implementing reliability as in equation 3-2.

3-1-3 Development horizon and computation time

In this section, the grid expansion optimization method is expanded to inspect a combination of multiple modifications as a group instead of individually. The options for extending this horizon are showcased, and the notion of computational cost is discussed.

3-1-3-1 Expansion horizon

In optimizing the grid, the current scheme optimizes the grid one modification at a time. In doing so, it effectively searches for a local minimum with the starting grid as the initial point. The modification resulting from optimizing the grid expansion for one improvement at a time may, however, not be part of the optimal modifications when optimizing for more modifications at a time, i.e. the global minimum.

We thus want to explore the effect of looking at different combinations of modifications, and implement the one with the eventually better performance. In other words, sacrificing short-term gains for unlocking longer-term improvements. We introduce the parameter for branch depth D. D is the number of modifications tested for at once or a measure of the size of combinations of modifications; The *planning horizon*.

This way of lengthening the planning horizon however, allows for little control over computation time. Given a relatively small grid of 10 nodes, lengthening the planning horizon by 1 modification increases the computation time 45-fold (we will get into the specific calculation in the next section).

We introduce a second measure defining the horizon, *breadth* B. For each increase in depth, we only inspect the modifications corresponding to the B best branches of the previous planning step. This results in a tree structure, where only the "child" modifications of the B most promising initial ones are explored.

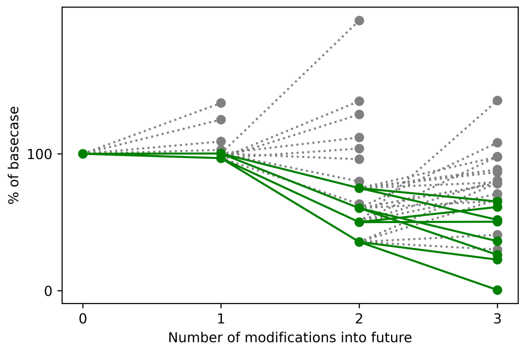

Figure 3-2: Illustration of grid expansion with horizon depth $D = 3$, the number of possible modifications 6 and $B = 2$. Gray dashed lines correspond to explored modifications which were pruned, green lines correspond to explored and most promising branches U_{branch}.

This approach allows us to fine-tune the computation time by selecting appropriate values for D and B. For a B value equal to the number of possible modifications, the new algorithm corresponds to a full Branch-and-Bound algorithm with horizon D.

3-1-3-2 Time complexity of grid design

Because we design our grid expansion program to run in a reasonable time frame, it is useful to find a notion of time complexity of the program. In this section, we will quantify this as the number of optimal power flow computations N necessary to be ran for each completed modification.

The grid expansion problem is of combinatorial nature. Adding an extra node to a grid with n nodes also adds n extra possible connections to be modified. The computation time for this grid expansion explodes fairly quickly. The number of Optimal Power Flow optimizations for a grid expansion study given a grid with n nodes scales with

$$\mathsf{N}_{Greedy} = (\frac{n(n-1)}{2}) \tag{3-6}$$

If we would want to inspect two modifications at once, we would have to inspect $(n(n-1)/2)^2$, neglecting some duplicates. These duplicates arise when we don't consider two distinct combinations of modifications (u_1, u_2) and (u_2, u_1) to be functionally different, since they result in ultimately the same grid. In general the number of optimal power flow operations scales with

$$\mathsf{N}_{OPF} = \frac{(\frac{n(n-1)}{2} + D - 1)!}{D!(\frac{n(n-1)}{2} - 1)!} \tag{3-7}$$

$$\mathsf{N}_{OPF,dup} = (\frac{n(n-1)}{2})^D \tag{3-8}$$

N_{OPF} and $\mathsf{N}_{OPF,dup}$ are the number of Optimal Power Flow optimizations needed to be run when removing duplicate sets of modifications or not, respectively. These distributions correspond with the combination and permutation of D draws from $n(n-1)/2$ possible modifications, allowing for repetition of modification candidates.

Adding to the development horizon now increases the number of calculations needed drastically. For a grid of 10 nodes, inspecting two modifications at once increases the number of calculations needed 23- or 45-fold, depending on whether we discard duplicates or not.

Because this value of depth D alone allows for little fine control over the computation time, we introduces breadth parameter B. Now, since the number of duplicate end-branches that can be discarded heavily depends on which branches are chosen in the steps before, we can only find a best-case complexity when discarding duplicates. The number of operations now scale with

$$\mathsf{N}_{OPF}^{B} \geq \sum_{h=0}^{D-1} \left(\frac{n(n-1)}{2} \cdot \frac{(B+h-1)!}{h!(B-1)!} - \left(\frac{(B+h-1)!}{h!(B-1)!} B - \frac{(B+h)!}{(h+1)!(B-1)!} \right) \right) \tag{3-9}$$

$$\mathsf{N}_{OPF,dup}^{B} = \sum_{h=0}^{D-1} \left(\frac{n(n-1)}{2} \cdot B^h \right) \tag{3-10}$$

Where equation 3-9 is based on the given that at depth D, there are at least $\binom{B+D-1}{D}$ unique branches that need to optimized for each of the $\frac{n(n-1)}{2}$ subsequent modifications, with the resulting duplicates from that subsequent step subtracted from the number of modifications. The worst-case complexity now results in the situation that no branches are discarded, or equation 3-10. For the full derivation, see Appendix 6-1-2.

We can compare the number of operations needed between a full breadth optimization ($B = \frac{n(n-1)}{2}$) or a limited breadth, for different values of depth D.

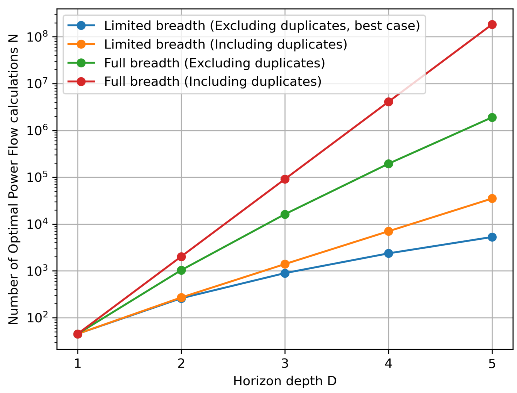

Figure 3-3: Number of Optimal Power Flow computations necessary for different horizons and for full breadth exploration and exploration with $B = 5$. Grid with 10 nodes.

Where we see that with a more narrow horizon, the number of required operations is a lot shorter. By tuning B and D we can strike a fine balance between exploration and computation time.

Even though removing duplicates could offer a considerable computational advantage, we will still use the approach of computing duplicate values. This allows us to still differentiate between two modification sequences using the order of the modifications and compare expectations to actual computation times.

3-1-4 Cost function

In this section, the cost functions used for grid operation and expansion are designed. A set of two cost functions - for the Monte-Carlo approach and the scenario approach - for optimal power flow are introduced and a set of three grid expansion cost functions are described.

3-1-4-1 Optimal power flow

In this section, we will introduce the optimal power flow optimization functions used in this thesis. In general, the aim is to make the Monte-Carlo approach to grid operation and the scenario approach to grid operation have similar optimization functions, as well as both optimization functions be as simple as possible, as to optimize in reasonable time.

We adopt the power flow formula as in equation 2-25, and choose to limit this voltage level per node between a minimum and a maximum like in the first row of equation 2-26

$$|v_{\mathcal{L}}| = V_0 \mathbb{1} + \frac{1}{V_0}(Z_p(P_\delta + P_\theta) + Z_q(Q_\delta + Q_\theta))$$

$$V_{min} \leq |v_{\mathcal{L}}| \leq V_{max}$$

We constrain power as in equation 2-27.

$$\begin{cases} P_{min} \leq P_\theta \leq P_{max} & \text{Limit on active power} \\ Q_{min} \leq Q_\theta \leq Q_{max} & \text{Limit on reactive power} \end{cases}$$

We can then, by introducing extended vector S^*, rewrite equation 2-25 into a more standard form

$$\begin{aligned} V_0(|V_{min}| - V_0) - Z^*\delta^{(i)} \leq Z^* S^* &\leq V_0(|V_{max}| - V_0) - Z^*\delta^{(i)} \\ S^* &= \begin{bmatrix} P_\theta & Q_\theta \end{bmatrix}^T \\ \delta^{(i)} &= \begin{bmatrix} P_\delta^{(i)} & Q_\delta^{(i)} \end{bmatrix}^T \\ Z^* &= \begin{bmatrix} Z_P & Z_Q \end{bmatrix} \end{aligned}$$

Where we now constructed a convex constraint on S^* of standard expression $Ax \leq b$. We can insert this constraint into the optimization function for both a scenario approach optimal power flow as a Monte-Carlo approach optimal power flow optimization model.

Scenario approach

The optimization function for using the scenario approach to optimal power flow we used is

$$
\begin{aligned}
\hat{f}_{\mathcal{G}}^{Sc} &= \min_{S^*} f(S^*) \\
S^* &= \begin{bmatrix} P_\theta & Q_\theta \end{bmatrix}^T \\
\\
S^*_{min} &\leq S^* \leq S^*_{max} \\
b_{min} &\leq Z^* S^* \leq b_{max} \\
\\
S^*_{min} &= \begin{bmatrix} P_{min} & Q_{min} \end{bmatrix}^T \\
S^*_{max} &= \begin{bmatrix} P_{max} & Q_{max} \end{bmatrix}^T \\
b_{min} &= \begin{bmatrix} \max_{i \in \{1,2,\cdots,N\}} b^{(i)}_{min,1} & \max_{i \in \{1,2,\cdots,N\}} b^{(i)}_{min,2} & \cdots & \max_{i \in \{1,2,\cdots,N\}} b^{(i)}_{min,n} \end{bmatrix}^T \\
b_{max} &= \begin{bmatrix} \min_{i \in \{1,2,\cdots,N\}} b^{(i)}_{max,1} & \min_{i \in \{1,2,\cdots,N\}} b^{(i)}_{max,2} & \cdots & \min_{i \in \{1,2,\cdots,N\}} b^{(i)}_{max,n} \end{bmatrix}^T \\
b^{(i)}_{min} &= V_0(|V_{min}| - V_0) - Z^* \delta^{(i)} \\
b^{(i)}_{max} &= V_0(|V_{max}| - V_0) - Z^* \delta^{(i)} \\
\\
Z^* &= \begin{bmatrix} Z_P & Z_Q \end{bmatrix} \\
\delta^{(i)} &= \begin{bmatrix} P^{(i)}_\delta & Q^{(i)}_\delta \end{bmatrix}^T \\
Z_P &= \text{Real}(Y)^{-1} \\
Z_Q &= \text{Imag}(Y)^{-1} \\
Y &= \begin{bmatrix} w_{sh,1} + \sum_Y w_{1,j} & -w_{1,2} & \cdots & -w_{1,N} \\ -w_{2,1} & w_{sh,2} + \sum_Y w_{2,j} & \cdots & -w_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ -w_{N,1} & -w_{N,2} & \cdots & w_{sh,N} + \sum_Y w_{N,j} \end{bmatrix} \\
w_{i,j} &= w_{j,i} \;\; \textit{The complex admittance between node } i \textit{ and } j \\
w_{sh,i} & \;\; \textit{The shunt admittance at node } i\textit{, assumed to be } 0
\end{aligned}
\tag{3-11}
$$

Where b_{min} and b_{max} represent the most limiting samples. This allows for fairly easily finding candidates for discarding scenarios if the a-posteriori ϵ allows, since all candidates will be an element of b_{min} or b_{max}.

Monte-Carlo approach

The optimization function for using the Monte-Carlo approach to optimal power flow we used is

$$
\begin{aligned}
\hat{f}_{\mathcal{G}}^{MC} = \frac{1}{N}\sum_{i=1}^{N} \min_{S^*} f(S^*) \\
S^* &= \begin{bmatrix} P_\theta & Q_\theta \end{bmatrix}^T \\
\\
S^*_{min} &\leq S^* \leq S^*_{max} \\
b^{(i)}_{min} &\leq Z^* S^* \leq b^{(i)}_{max} \\
\\
S^*_{min} &= \begin{bmatrix} P_{min} & Q_{min} \end{bmatrix}^T \\
S^*_{max} &= \begin{bmatrix} P_{max} & Q_{max} \end{bmatrix}^T \\
b^{(i)}_{min} &= V_0(|V_{min}| - V_0) - Z^* \delta^{(i)} \\
b^{(i)}_{max} &= V_0(|V_{max}| - V_0) - Z^* \delta^{(i)} \qquad (3\text{-}12) \\
\\
Z^* &= \begin{bmatrix} Z_P & Z_Q \end{bmatrix} \\
\delta^{(i)} &= \begin{bmatrix} P^{(i)}_\delta & Q^{(i)}_\delta \end{bmatrix}^T \\
Z_P &= \text{Real}(Y)^{-1} \\
Z_Q &= \text{Imag}(Y)^{-1} \\
Y &= \begin{bmatrix} w_{sh,1} + \sum_Y w_{1,j} & -w_{1,2} & \cdots & -w_{1,N} \\ -w_{2,1} & w_{sh,2} + \sum_Y w_{2,j} & \cdots & -w_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ -w_{N,1} & -w_{N,2} & \cdots & w_{sh,N} + \sum_Y w_{N,j} \end{bmatrix} \\
w_{i,j} &= w_{j,i} \textit{ The complex admittance between node } i \textit{ and } j \\
w_{sh,i} & \textit{ The shunt admittance at node } i\textit{, assumed to be } 0
\end{aligned}
$$

Which is similar to the scenario approach, with the only difference being that instead of optimizing over all samples simultaneously, this optimization function calculates the optimal value for all samples individually and averages the result.

3-1-4-2 Grid expansion

Solution space and general form

For grid expansion, we first consider the control space U. This control space is defined as the union of the set of all possible admittance modifications that can be applied to existing edges, and the set of all new edges possible to be added.

$$
\begin{aligned}
U &= U_{upg} \cup U_{add} \\
U_{upg} &= \{(v_i, v_j, w^+) | v_i, v_j \in \mathcal{V}, (v_i, v_j) \in \mathcal{E}, w^+ \in \mathbb{R}^+\} \\
U_{add} &= \{(v_i, v_j, w) | v_i, v_j \in \mathcal{V}, (v_i, v_j) \notin \mathcal{E}, w \in \mathbb{R}^+\}
\end{aligned}
$$

Where w^+ is the admittance upgrade to be installed on an existing edge, and w is the admittance of the newly added edge. For simplicity, we choose w^+ and w to be equal and fixed for a single grid expansion optimization.

We opt to implement a greedy optimization algorithm, which takes existing graph $\mathcal{G}$ and chooses the best single modification from set U. In general, we describe the expansion program as

$$\min_{u \in U_{upg} \cup U_{add}} g(\mathcal{G}, u) \tag{3-13}$$

Where $g(\mathcal{G}, u)$ denotes a function for testing the performance improvement of graph $\mathcal{G}$ after modification u. This function is specific to the optimization approach. In general, we use three variables in this optimization:

- An improvement in terms of reliability
- An improvement in terms of operational performance
- A cost associated with the modification

As U is a discrete set, we have no choice but to loop over all possibilities in U, and choose the best. As discussed before, this results in a total of $n(n-1)$, with n the number of nodes, possibilities being studied per iteration.

Scenario approach

For the scenario approach, we use the following function $g_1^{Sc}(\mathcal{G}, u)$.

$$\begin{aligned} &\min_{u \in U_{upg} \cup U_{add}} g_1^{Sc}(\mathcal{G}, u) \\ &g_1^{Sc}(\mathcal{G}, u) = W \begin{bmatrix} 1 - 2\mathbb{P}^N\{V_{\mathcal{G}} \geq V_{\mathcal{G}+u}\} \\ \frac{\hat{f}^{Sc}_{\mathcal{G}+u} - \hat{f}^{Sc}_{\mathcal{G}}}{\hat{f}^{Sc}_{\mathcal{G}}} \\ d(u) \end{bmatrix} \Bigg|_{\delta} \end{aligned} \tag{3-14}$$

Which is a weighted sum with weight vector $W \in \mathbb{R}^4$ of the three variables considered.

The first term is calculated using equation 3-2 using the complexities $k_{\mathcal{G}}$ and $k_{\mathcal{G}+u}$, normalized to be in range $[-1, 1]$.

The second term is a relative improvement in performance level using the scenario approach $\hat{f}^{Sc}_{\mathcal{G}}$, calculated using optimization equation 3-11.

The third term is a function of the euclidean distance covered by the modification and the type of modification, scaled such that the maximum value is 1, or $\{d(u) : u \in U\} \subseteq (0, 1]$. The type of modification, $u \in U_{upg}$ or $u \in U_{add}$ determines which cost weight is used, W_3 and W_4 respectively. The formal definition is as in equation 3-15.

$$d(u) := \begin{cases} \begin{bmatrix} \|p_2 - p_1\|_2 \\ 0 \end{bmatrix} & u \in U_{add} \\ \begin{bmatrix} 0 \\ \|p_2 - p_1\|_2 \end{bmatrix} & u \in U_{upg} \end{cases} \tag{3-15}$$

With p_1 and p_2 the positions of the two nodes connected by u.

Monte-Carlo approach

For the Monte-Carlo approach, we use the following function $g_1^{MC}(\mathcal{G}, u)$.

$$\begin{aligned} &\min_{u \in U_{upg} \cup U_{add}} g_1^{MC}(\mathcal{G}, u) \\ &g_1^{MC}(\mathcal{G}, u) = W \begin{bmatrix} \frac{\mathbb{P}\{V_{\mathcal{G}+u}\} - \mathbb{P}\{V_{\mathcal{G}}\}}{\mathbb{P}\{V_{\mathcal{G}}\}} \\ \frac{\hat{f}^{MC}_{\mathcal{G}+u} - \hat{f}^{MC}_{\mathcal{G}}}{\hat{f}^{MC}_{\mathcal{G}}} \\ d(u) \end{bmatrix} \Bigg|_{\delta} \end{aligned} \tag{3-16}$$

Which is a weighted sum with weight vector $W \in \mathbb{R}^4$ of the three variables considered.

The first term is calculated as the relative improvement in the fraction of infeasible samples, calculated as the fraction of samples that have no feasible solution in equation 3-12.

The second term is a relative improvement in performance level using the scenario approach $\hat{f}^{MC}_{\mathcal{G}}$, calculated using optimization equation 3-12.

The third term is the same function as in equation 3-14, namely equation 3-15.

Longer development horizon

For a longer development horizon, we define the sequence of collectively investigated modifications $U_{branch} = (u_1, \ldots, u_h) \in U$. We then define the cost function $g_h^{Sc}(\mathcal{G}, (u_1, \ldots, u_h))$.

$$
\begin{aligned}
&\min_{(u_1,\ldots,u_h)\in U} g_h^{Sc}(\mathcal{G}, (u_1, \ldots, u_h)) \\
&g_h^{Sc}(\mathcal{G}, (u_1, \ldots, u_h)) = W \begin{bmatrix} 1 - 2\mathbb{P}^N\{V_{\mathcal{G}} \geq V_{\mathcal{G}+u_1+\cdots+u_h}\} \\ \frac{\hat{f}^{Sc}_{\mathcal{G}+u_1+\cdots+u_h} - \hat{f}^{Sc}_{\mathcal{G}}}{\hat{f}^{Sc}_{\mathcal{G}}} \\ \sum_{u\in\{u_1,\ldots,u_h\}} d(u) \end{bmatrix}\Bigg|_{\delta}
\end{aligned} \tag{3-17}
$$

Where we can calculate the improvement against the starting graph along each step of the development horizon for $h = 1, \ldots, D$.

3-1-5 Optimization loop

In this section, the entirety of the optimization program is showcased, starting with the stopping criteria. Second, the optimization loop used in the simulations is described and shown as a process diagram.

3-1-5-1 Stopping criteria

As it currently stands, the optimization has no stopping criterion yet. In this section, we introduce the two stopping criteria used for the optimization loop.

Insufficient improvement
The first stopping criterion acts on the result of the cost function. It is specified as

$$\begin{aligned} &g((G), \hat{u}) > c_{mod} \\ &\qquad \hat{u} = \arg\min_{u_c \in U} g(\mathcal{G}, u_c) \text{ The optimal modification on graph } \mathcal{G} \end{aligned} \tag{3-18}$$

Thus ending the optimization run if the optimal modification, and as a consequence any possible modification, does not yield enough of an improvement.

Total budget
The second stopping criterion acts as a budget for the total cost of installed modifications. It is specified as

$$\begin{aligned} &\sum_{\hat{u} \in \hat{U}} W_{3,4} d(\hat{u}) > c_{tot} \\ &\qquad \hat{U} = (\hat{u}_1, \hat{u}_2, \dots) \text{ The sequence of installed modifications} \end{aligned} \tag{3-19}$$

Endig the simulation run if a sufficient level of capital is expended. It acts as a limiter on the number of modifications that can be added in a single simulation run.

In the optimization loop, the candidate modifications that would trigger this stopping criterion are filtered out and not considered in finding the optimal modification, allowing the optimization run to fully run out the budget as long as all modifications within the remainder of the budget do not trigger the stopping criterion defined in equation 3-18.

3-1-5-2 Greedy approach

Since we aim to compare the Monte-Carlo method and the scenario, we design the optimization scheme such that both methods use the same fundamental optimization loop. We describe this loop as follows:

1. Specify graph $\mathcal{G}$ and $\epsilon \in (0,1)$ and $\beta \in (0,1)$ for the level and confidence parameter.
2. Use ϵ and β to find N_{log} the lower limit on samples needed to conclude ϵ and β.
3. Pick $N \geq N_{log}$ as the number of samples.
4. Generate $\{\delta^1, \delta^2, \cdots, \delta^N\} \in \Delta^N$ as i.i.d samples for our active and reactive sampled loads scenarios.
5. Compute $\hat{f}_{\mathcal{G}}^{MC}$ and $\mathbb{P}\{V_{\mathcal{G}}\}$, or $\hat{f}_{\mathcal{G}}^{Sc}$ and $k_{\mathcal{G}}$ as described in equations 3-12, 3-11 and the discarding procedure in section 2-2-7-2.
6. Compute U as a union of U_{upg} and U_{add}.
7. For each $u_c \in U$, compute the following
   i Apply modification u_c to graph $\mathcal{G}$ and calculate $d(u_c)$.
   ii Compute admittance matrix Y as the weighted Laplacian of graph $[\mathcal{G} + u_c]$ and invert it for matrices Z_P and Z_Q.
   iii Compute $\hat{f}_{\mathcal{G}+u_c}^{MC}$ and $\mathbb{P}\{V_{\mathcal{G}+u_c}\}$, or $\hat{f}_{\mathcal{G}+u_c}^{Sc}$ and $k_{\mathcal{G}+u_c}$ as described in equations 3-12, 3-11 and the discarding procedure in section 2-2-7-2.
   iv Compute $g(\mathcal{G}, u_c)$.
8. Choose the best modification $\hat{u} = \arg\min_{u_c \in U} g(\mathcal{G}, u_c)$ and implement it, calculate new values for Y, Z_P and Z_Q, and return to step 5. Stop if any of the stopping criteria have been met.

We can graphically illustrate this optimization loop as

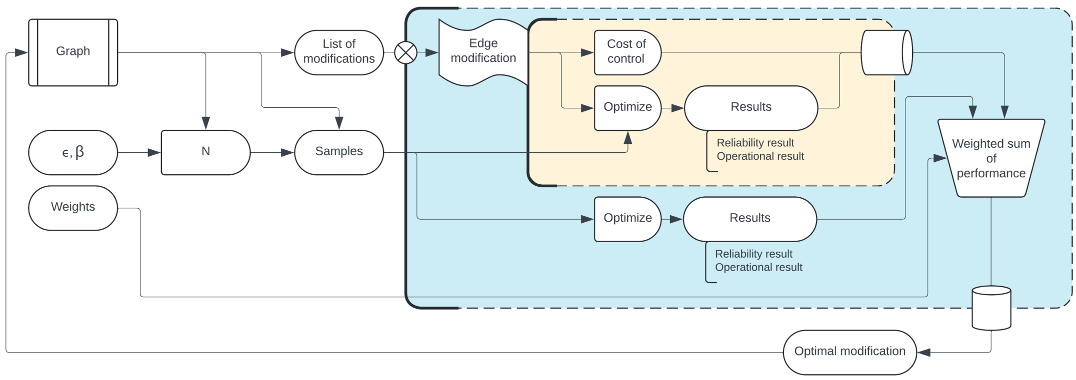

Figure 3-4: Expansion loop for the greedy approach.

3-1-5-3 Development horizon

As discussed in section 3-1-3-1, we implement a tree traversal algorithm where only the B most optimal modifications of each branch are explored further, up to depth D. This makes the algorithm more complicated, but the main optimization mechanics remain unchanged. We describe the optimization loop to that end as follows:

1. Specify graph $\mathcal{G}$ and $\epsilon \in (0,1)$ and $\beta \in (0,1)$ for the level and confidence parameter. Define parameters D and B.
2. Use ϵ and β to find N_{log} the lower limit on samples needed to conclude ϵ and β.
3. Pick $N \geq N_{log}$ as the number of samples.
4. Generate $\{\delta^1, \delta^2, \cdots, \delta^N\} \in \Delta^N$ as i.i.d samples for our active and reactive sampled loads scenarios.
5. Repeat the following, D times. Initalize modification sequence as empty sequence $U_{branch} = ()$.
   (a) Compute $\hat{f}^{Sc}_{\mathcal{G}}$ and $k_{\mathcal{G}}$ as described in equations 3-12, 3-11 and the discarding procedure in section 2-2-7-2.
   (b) Compute U as a union of U_{upg} and U_{add}.
   (c) For each candidate $u_c \in U$, compute the following
      i Apply modification u_c to graph $[\mathcal{G}+\sum_{u\in U_{branch}} u]$ and calculate $d(u_c)+\sum_{u\in U_{branch}} d(u)$.
      ii Compute admittance matrix Y as the weighted Laplacian of graph $[\mathcal{G} + u_c + \sum_{u\in U_{branch}} u]$ and invert it for matrices Z_P and Z_Q.
      iii Compute $\hat{f}^{Sc}_{\mathcal{G}+u_c+\sum_{u\in U_{branch}} u}$ and $k_{\mathcal{G}+u_c+\sum_{u\in U_{branch}} u}$ as described in equations 3-11 and the discarding procedure in section 2-2-7-2.
      iv Compute $g^{Sc}_h(\mathcal{G}, (U_{branch}, u_c))$ with h the current depth.
   (d) Find the set of the B most optimal modifications, excluding those that would meet any stopping criterion.
   $\{u_c \in U : |\{g^{Sc}_h(\mathcal{G}, (U_{branch}, u)), u \in U\} \cap (-\infty, g^{Sc}_h(\mathcal{G}, (U_{branch}, u_c)))]| \leq B\}$
   (e) For each candidate modification u_c in this set, copy graph G and the current modification sequence U_{branch}, implement this u_c in the copied graph and add it to the copied sequence. Return to step (a).

   We now have a set of all investigated modification sequences, or explored subtree $T^B_D = \{U^{1,1,\ldots,1}_{branch}, U^{1,1,\ldots,2}_{branch}, \ldots, U^{B,B,\ldots,B}_{branch}\}$, with the superscript denoting the indexing of branches taken. $|T^B_D| = B^D$.
6. Find the modification with the best value at the end of all possible resulting sequences using $\hat{U}_{branch} = \arg\min_{U_{branch}\in T^B_D} g^{Sc}_h(\mathcal{G}, U_{branch})$, and implement $(\hat{U}_{branch})_1$ in graph G. If two or more modifications lead to the same ultimate result, implement the one with the best short-term score $g(\mathcal{G}, (\hat{U}_{branch})_1)$.
7. Calculate new values for Y, Z_P and Z_Q, and return to step 5. Stop if any of the stopping criteria have been met.

We can graphically illustrate this optimization loop as

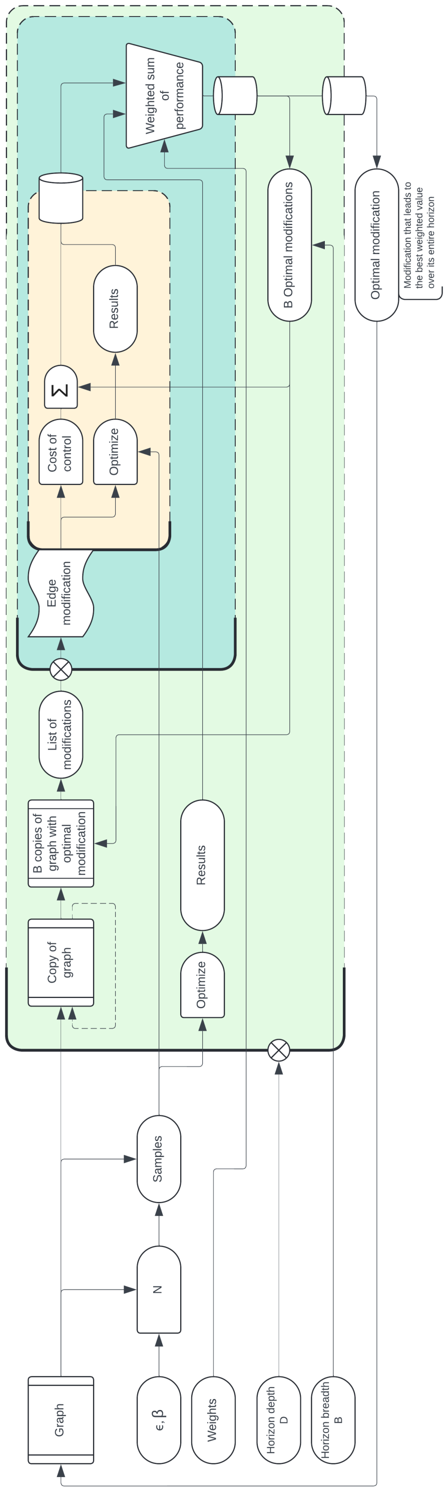

Figure 3-5: Expansion loop for the approach with extended horizon

3-2 Validation of results

After the grid expansion optimization has found the suggested modification sequences, we want to compare them against sequences suggested using other parameters. In this section, we describe this process

3-2-1 Performance

We compare operational performance of the graph during all modification sequences. We want to do this in two ways: using the scenario approach, since some grid operation studies are currently using that approach to grid operation, and the Monte-Carlo approach since current grid expansion studies' programs also use this method to gauge results. The method of getting these results is showcased in this section.

3-2-1-1 Scenario approach

In order to compare the results of optimization runs, we test improvements in terms of the optimal power flow cost $\hat{f}_{\mathcal{G}}^{Sc}$ using the scenario approach. This is done as this approach is most relevant in current robust grid operation studies. The optimal power flow cost can be computed using equation 3-11 and the procedure described in section 2-2-7-2. These results will be computed using the same sample set $\{\delta^1, \delta^2, \cdots, \delta^N\} \in \Delta^N$ as the set used in the simulation run.

The results for $\hat{f}_{\mathcal{G}}^{Sc}$ are then compared to the cumulative capital expenditure required for these improvements. This is calculated as

$$\sum_{\hat{u} \in \hat{U}} W_{3,4} d(\hat{u}) \qquad \hat{U} = (\hat{u}_1, \hat{u}_2, \dots) \text{ The sequence of installed modifications} \tag{3-20}$$

3-2-1-2 Monte-Carlo approach

To compare the grid expansion program with the scenario approach and the grid expansion program with the Monte-Carlo approach, we also compute the Monte-Carlo approach grid operation cost $\hat{f}_{\mathcal{G}}^{MC}$ using equation 3-12. This is done to compare the new expansion approach to the current standard.The results are computed using a novel sample set of $N_{MC} \gg N_{log}$ samples

The results for $\hat{f}_{\mathcal{G}}^{MC}$ are then compared to the cumulative capital expenditure required for these improvements, calculated using equation 3-20.

3-2-2 Reliability

As we want ot explicitly find reliability improvements, we measure reliability in two ways: using the scenario approach, to check if the same reliability claims hold and/or improve, and the Monte-Carlo approach since current grid expansion studies' programs use this method. The procedure of getting these results is showcased in this section.

3-2-2-1 Scenario approach

To test robustness using the scenario approach, we go back to the original definition of the violation probability in equation 2-4:

$$V(\theta) \doteq \text{Prob}\{\delta \in \Delta : \theta \notin \Theta^{(\delta)}\}$$

To test the violation probability in the scenario optimization sense, we calculate an optimal input $\hat{\theta}$ using a random sample set $\{\delta^1, \delta^2, \cdots, \delta^N\} \in \Delta^N$ with N the same as during expansion optimization, using equation 3-11, and empirically test the probability that this sample is infeasible for a novel sample δ^{N+1}. This fraction gives a out of sample guarantee.

This violation probability is then compared to the cumulative capital expenditure required for the modifications calculated using equation 3-20. To achieve somewhat consistent results, the operation described above is repeated m times and averaged, and the violation probability of each optimal input $\hat{\theta}$ is empirically determined using N_{novel} novel random samples.

3-2-2-2 Monte-Carlo approach

To test robustness in the Monte-Carlo sense, we look for the fraction of samples than are of violation. Or the fraction of samples for which equation 3-12 has no feasible solution. This violation probability will be calculated during the procedure in section 3-2-1-2, with the same sample set of N_{MC} samples.

We again compare these values with the cumulative capital expenditure calculated using equation 3-20.

3-2-3 Computation time

Finally, we want to compare computation time between different optimization methods. This is computation time is measured as the expansion program runs, and shown as time per implemented modification.

The simulations were run on a i7-7700HQ CPU at 2.80 GHz with 16 GB of RAM. Minimal other processes were running on the same computer during grid expansion optimization in order to keep results consistent.

3-3 Case studies

In this section, we describe the four case studies used for this thesis, the standard parameters and cost functions used and introduce the three parameter studies run.

3-3-1 Initial grids

We ran parameter studies using 4 initial grids as shown in figure 3-6. In general, all grids consist of a mix of controllable power sources, passive network nodes, and sampled sources and sampled drains. Nodes with sampled power loads generally have a controllable part of $\pm 5\%$ or the expected load as a form of demand response.

As our chosen power flow model is symmetric between active and reactive power, we opt to test purely real power loads only for our case studies. For the full setup of the initial grids, see appendix 6-2.

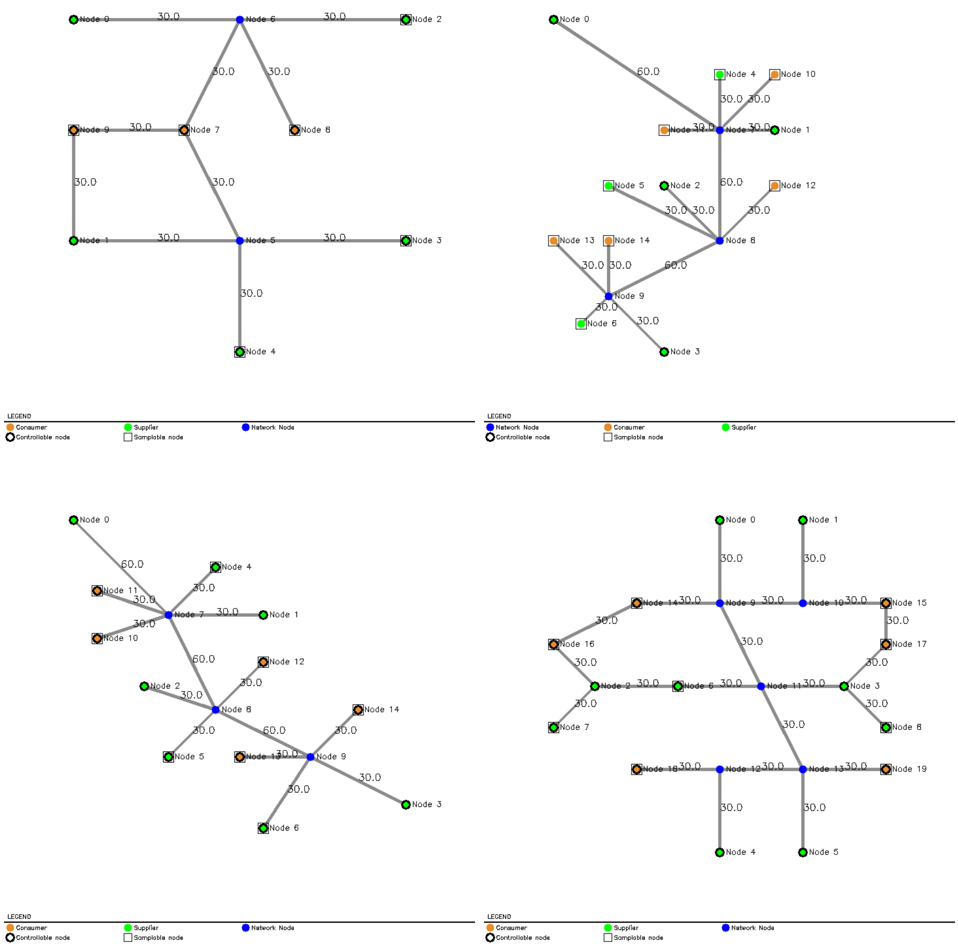

Figure 3-6: Initial grids used in the grid expansion program. Sizes range from 10 nodes (upper left) to 15 nodes (upper right and lower left) to 20 nodes (lower right). The full setup of this grids can be found in appendix 6-2

3-3-2 Standard parameters

For all simulation studies, a collection of parameters is used. To find the dependence on one of these parameters, all others have been kept constant within a single parameter study.

For all parameter studies except the first one, the scenario approach was used.

3-3-2-1 Optimal power flow

The standard values for the optimal power flow computation can be found in table 3-1. These values were chosen to be similar to existing optimal power flow studies [9], and the cost function was chosen as a simple convex function to aid with optimization.

Optimal power flow parameters		
Cost function	$f(S^*)$	$\begin{bmatrix}1 & \cdots & 1\end{bmatrix} S^*$
Level parameter	ϵ	0.05
Confidence parameter	β	10^{-5}
Baseline voltage level	V_0	1

Table 3-1: Standard values for optimal power flow parameters used in grid expansion studies

3-3-2-2 Grid expansion

The standard values for the grid expansion algorithm are shown in table 3-2. The values in W are chosen as to promote optimization over robustness and performance. D is set at 1 as the standard greedy approach has horizon 1, and setting B at 4 strikes a balance between computation time and exploration of the solution space U^D. Both stopping criteria were chosen to be lenient as to let the grid expansion program run its course and not prematurely terminate the run.

Grid expansion parameters		
Weight vector	W	$\begin{bmatrix}2000 & 1000 & 10 & 5\end{bmatrix}$
Capacity of added power line	w	30
Capacity upgrade for upgrading power lines	w^+	30
Branch depth	D	1
Branch breadth	B	4
Stopping criterion for individual modification	c_{mod}	5
Total budget	c_{tot}	20

Table 3-2: Standard values for grid expansion parameters used in grid expansion studies

3-3-2-3 Validation stage

The sample sizes and number of repeats in the validation stage are shown in table 3-3. These numbers were chosen at a level where results were consistent.

Validation stage parameters		
Number of samples for Monte-Carlo validation	N_{MC}	100000
Repeats of empirical violation probability study	m	50 [1]
Number of novel samples for empirical violation probability study	N_{novel}	100000

Table 3-3: Standard values for validation stage parameters used in grid expansion studies

3-3-3 Parameter studies

In this thesis, three parameter studies are presented. The parameters and their values are described in this section. All other parameters not mentioned in the study are equal to those in the previous section.

3-3-3-1 Optimization approach

In order to find if applying the scenario approach to grid expansion yields acceptable results, we compare it with the Monte-Carlo method. We run a parameter study with approaches:

- Monte-Carlo approach to optimal power flow and grid expansion
- Scenario approach to optimal power flow and grid expansion

3-3-3-2 Optimizing over robustness

In order to find if exploiting the evolution of the complexity of the grid with adding modifications yields improved results, we run the following parameter study:

- $W_1 = 2000$
- $W_1 = 0$

3-3-3-3 Branch depth

In order to find if lengthening the development horizon improves results, the following parameter study is run:

- $D = 1$
- $D = 2$
- $D = 3$

[1] For the largest graph of 20 nodes, a set of 10 samples is used as finding a feasible sample set is time intensive.

Chapter 4

Results

4-1 Optimization approach

In the first parameter study, we compare the performance between the two optimization approaches; Monte-Carlo and scenario.

We run 5 optimization runs for each setting, and compute the results in validation stage for each simulation run, and the average result for each setting.

4-1-1 Operational performance

In figure 4-1, we can see the operational result for both the Monte-Carlo approach and the scenario approach to grid expansion as well as grid operation. When evaluating the results, we see that for both the Monte-Carlo operational result as well as the scenario operational result the proposed modifications using the scenario expansion approach achieve similar performance or even outperform those proposed using the Monte-Carlo approach.

Results in terms of operational performance are computed using the procedures in sections 3-2-1-1 and 3-2-1-2.

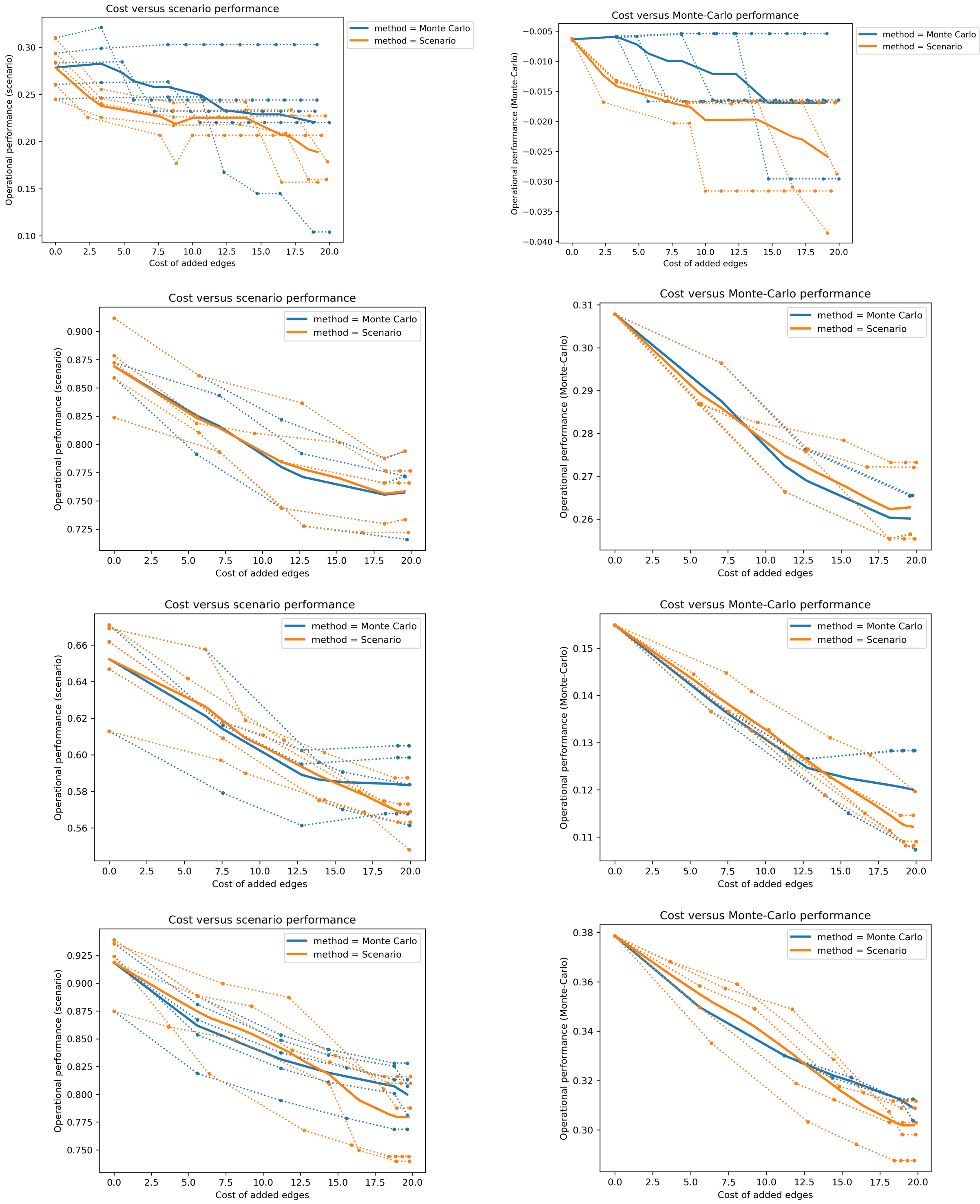

Figure 4-1: Operational performance versus modification cost both using the scenario approach (left) as well as the Monte-Carlo approach (right). Each row corresponds to an initial grid setting. Individual runs are shown as dashed lines, rolling averages as solid lines.

4-1-2 Reliability

In figure 4-3, we can see the reliability for both the Monte-Carlo approach and the scenario approach to grid expansion as well as grid operation. When evaluating the results, we see that for the violation probability the proposed modifications using the scenario expansion approach perform similar to those proposed using the Monte-Carlo approach, both leading to a grid that is more robust. The out of sample guarantees broadly tell the same story, except for the first initial graph, which seems to be an outlier.

Results in terms of violation probability are computed using the procedures in sections 3-2-2-1 and 3-2-2-2.

4-1-3 Computation time

In figure 4-2, we can see that the new computation method outperforms the Monte-Carlo method comfortably. The average time required per modification for this simulation was around a factor 20 shorter when using the scenario approach over the Monte-Carlo approach. For the largest grid, this factor was around 26. This speed improvement is a consequence of the scenario approach only having to run a couple optimizations (since we are discarding scenarios) where the Monte-Carlo approach has to optimize for all samples individually.

Results in computation time are measured using the procedure in section 3-2-3.

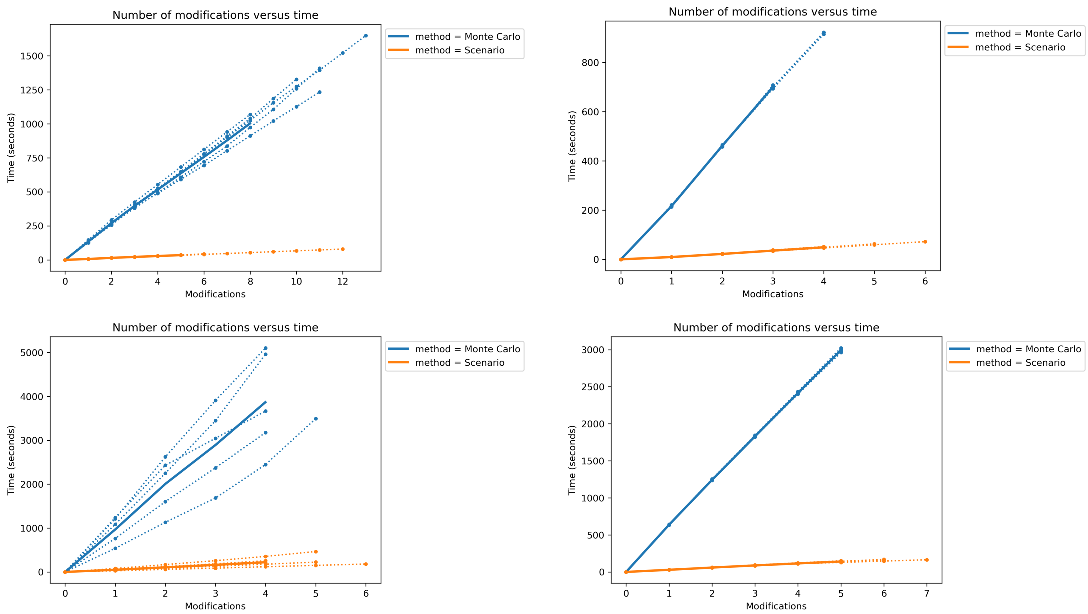

Figure 4-2: Computation time versus the number of installed modifications. The runs are ordered as: top left: Setup 1, top right: Setup 2, bottom left: Setup 3, bottom right: Setup 4. Individual runs are shown as dashed lines, rolling averages as solid lines.

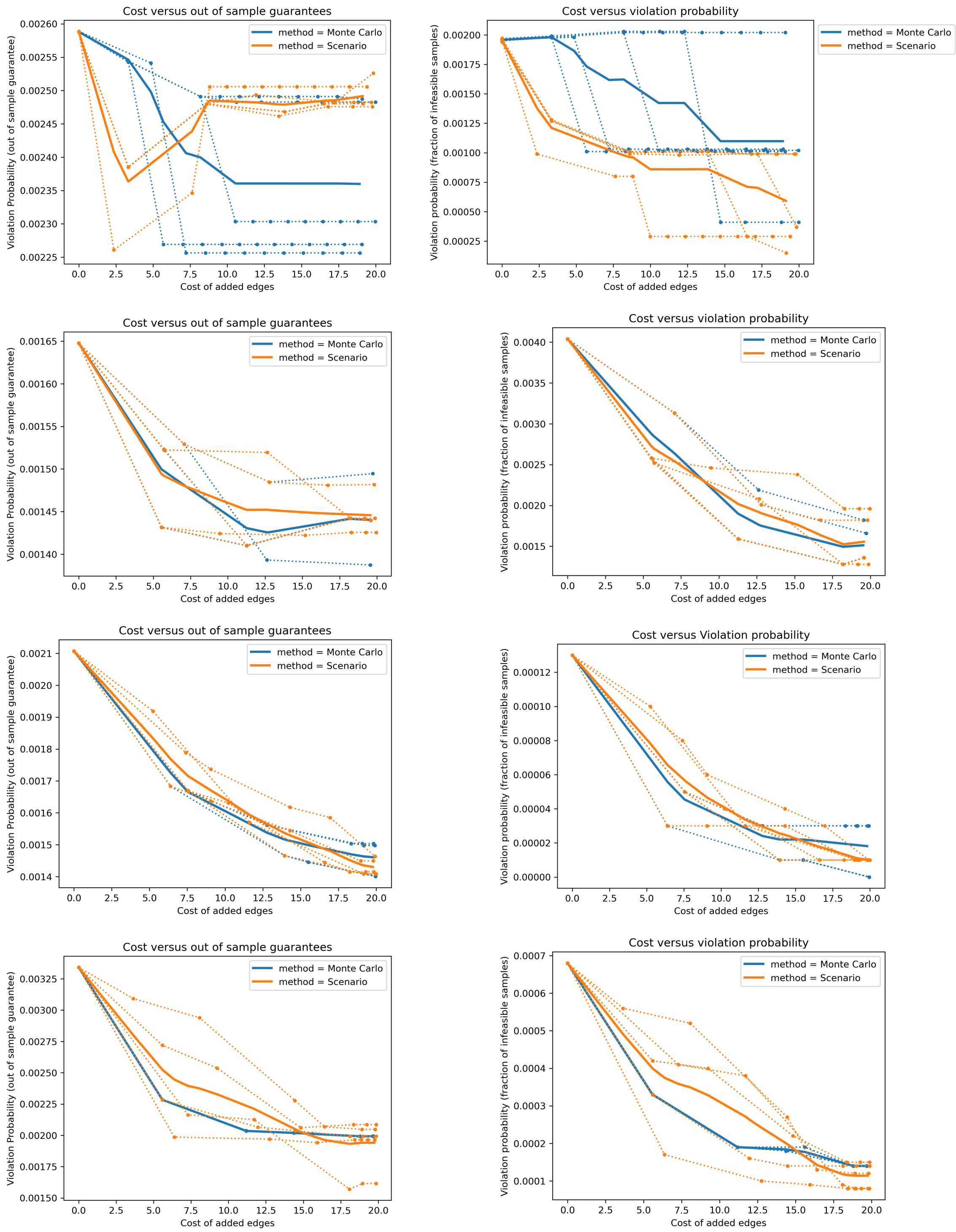

Figure 4-3: Violation probability versus modification cost both using the scenario approach (left) as well as the Monte-Carlo approach (right). Each row corresponds to an initial grid setting. Individual runs are shown as dashed lines, rolling averages as solid lines.

4-2 Optimizing over complexity

In the second parameter study, we compare the performance between optimizing explicitly over the number of support constraints and not considering this complexity explicitly.

We run 5 optimization runs for each setting, and compute the results in validation stage for each simulation run, and the average result for each setting.

4-2-1 Operational performance

In figure 4-4, we can see the operational result for both the Monte-Carlo approach and the scenario approach to grid expansion as well as grid operation. We see that not optimizing over the number of support constraints actually yields better results than not optimizing explicitly over the complexity of the optimal power flow optimization.

Results in terms of operational performance are computed using the procedures in sections 3-2-1-1 and 3-2-1-2.

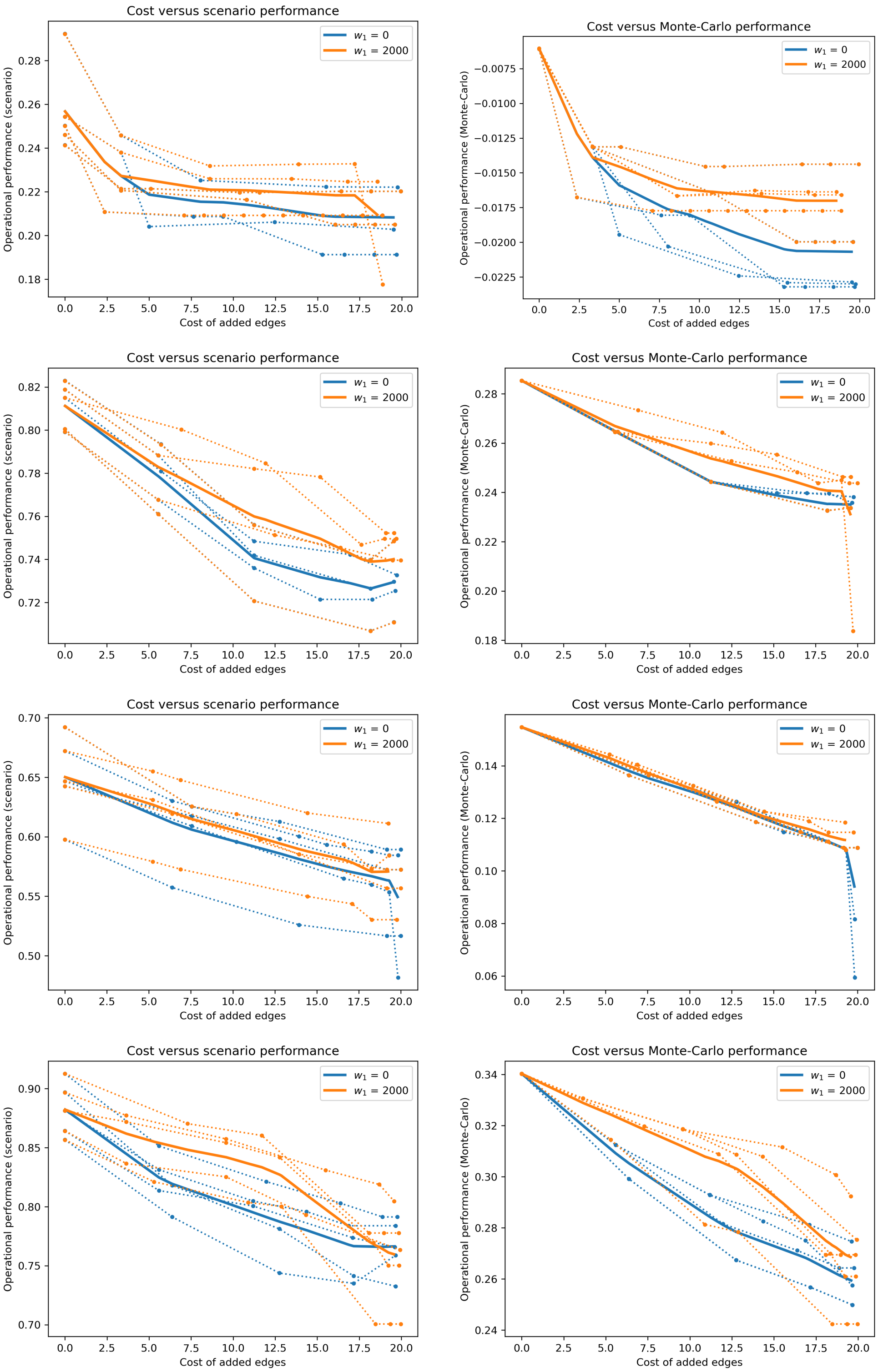

Figure 4-4: Operational performance versus modification cost both using the scenario approach (left) as well as the Monte-Carlo approach (right). Each row corresponds to an initial grid setting. Individual runs are shown as dashed lines, rolling averages as solid lines.

4-2-2 Reliability

In figure 4-6 we see this result hold: Optimizing not using the number of support constraints actually improves reliability of the grid. Near the end of the optimization, the results converge between the two settings.

Results in terms of violation probability are computed using the procedures in sections 3-2-2-1 and 3-2-2-2.

4-2-3 Computation time

In figure 4-5 we see that changing optimization weight vector W does not change the computation time needed, which is what we would expect.

Results in computation time are measured using the procedure in section 3-2-3.

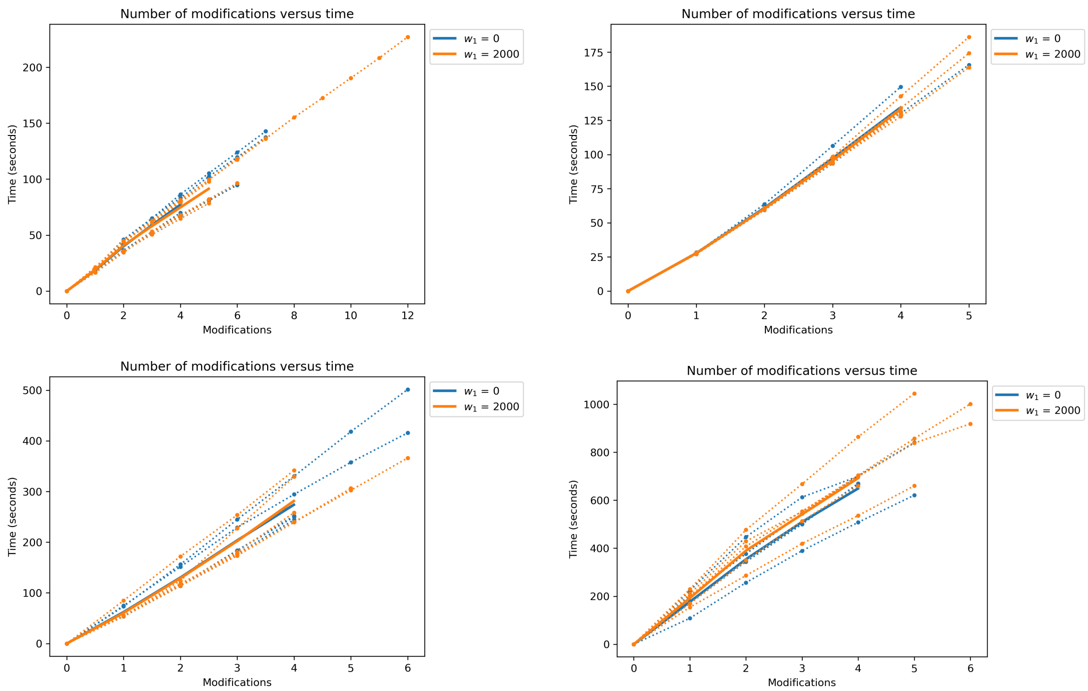

Figure 4-5: Computation time versus the number of installed modifications. The runs are ordered as: top left: Setup 1, top right: Setup 2, bottom left: Setup 3, bottom right: Setup 4. Individual runs are shown as dashed lines, rolling averages as solid lines.

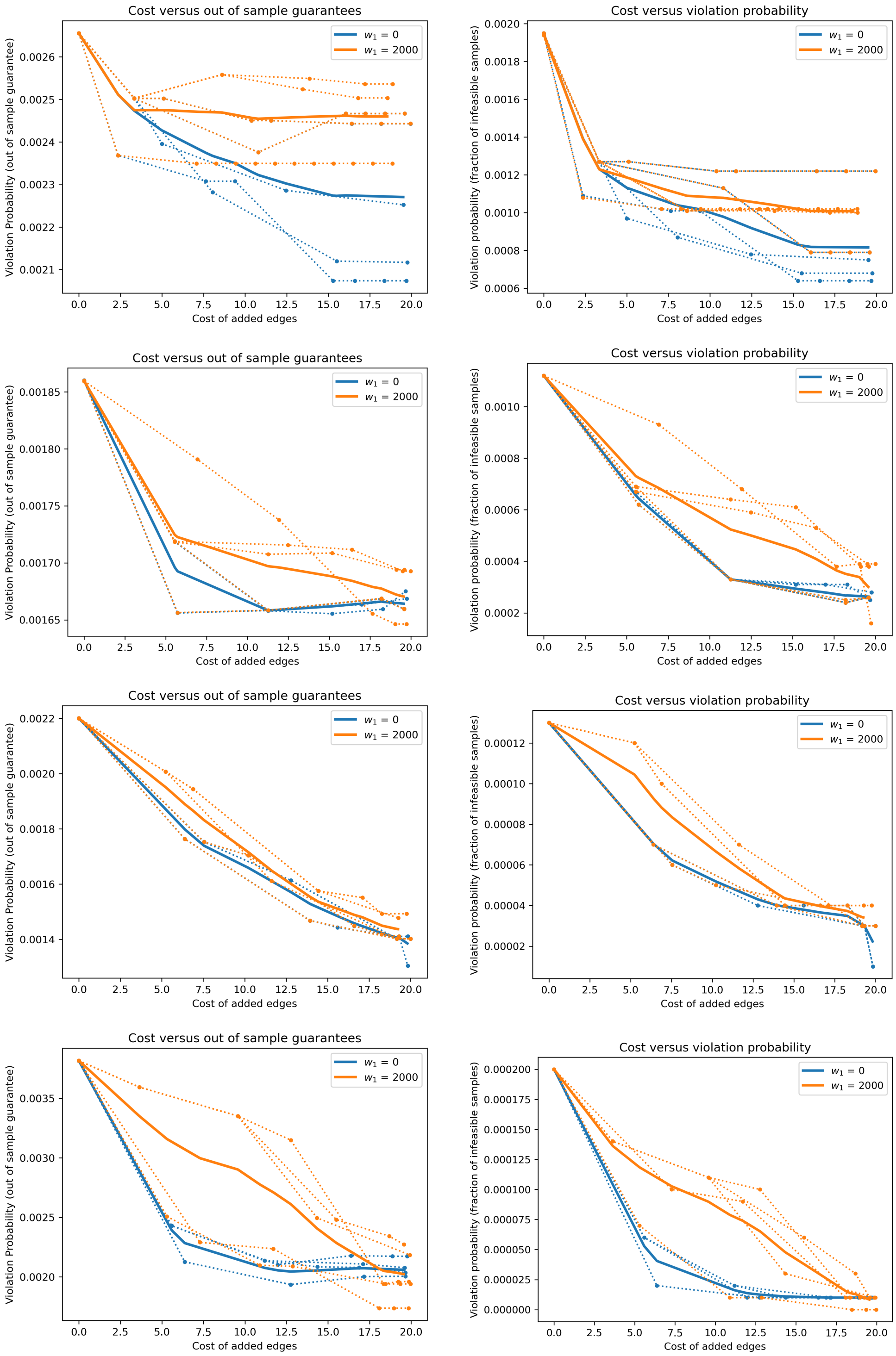

Figure 4-6: Violation probability versus modification cost both using the scenario approach (left) as well as the Monte-Carlo approach (right). Each row corresponds to an initial grid setting. Individual runs are shown as dashed lines, rolling averages as solid lines.

4-3 Branch depth

In the second parameter study, we compare the performance when optimizing over a largest development horizon, or different values of D.

We run 5 optimization runs for each setting, and compute the results in validation stage for each simulation run, and the average result for each setting.

4-3-1 Operational performance

In figure 4-7, we see the operational result for both the Monte-Carlo approach and the scenario approach to grid expansion as well as grid operation. We see that expanding the development horizon only yields improved results for larger graphs, but even then with diminishing returns. This could be a result of a breadth value B chosen such that computation times were manageable, limiting exploration.

Results in terms of operational performance are computed using the procedures in sections 3-2-1-1 and 3-2-1-2.

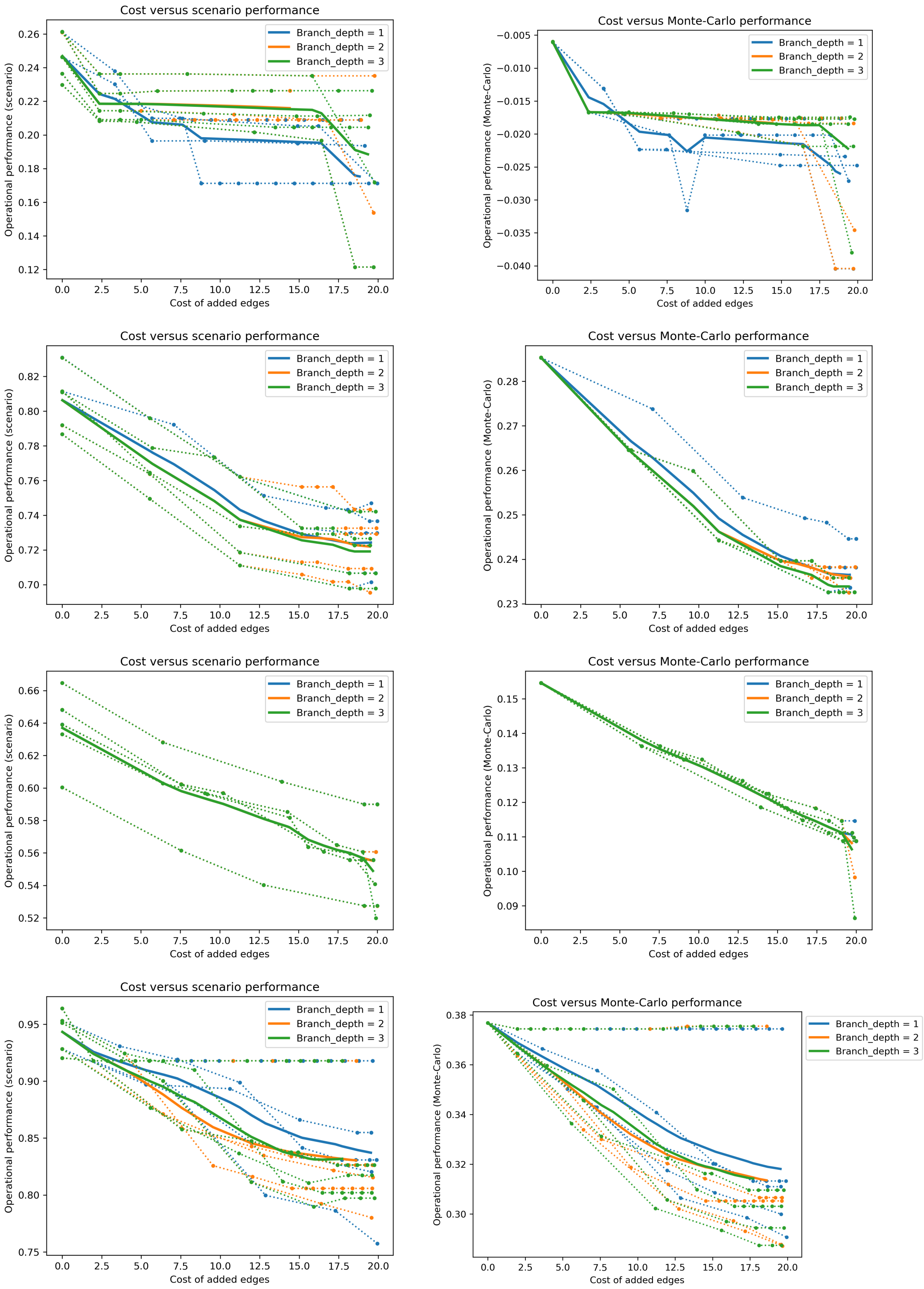

Figure 4-7: Operational performance versus modification cost both using the scenario approach (left) as well as the Monte-Carlo approach (right). Each row corresponds to an initial grid setting. Individual runs are shown as dashed lines, rolling averages as solid lines.

4-3-2 Reliability

In figure 4-9 the same is true, the simulation runs with $D = 2, 3$ only outperform the one with $D = 1$ for larger graphs, but are quite similar themselves.

Results in terms of violation probability are computed using the procedures in sections 3-2-2-1 and 3-2-2-2.

4-3-3 Computation time

In figure 4-8 we see that indeed, for a larger development horizon, the time required increases also. The required time needed per modification increases with ratio $1 : 5 : 21$ for $D = 1, 2, 3$, roughly in line with the expected $1 : 4 : 20$ calculated using 3-10.

A second notable behavior is that for a longer horizon, a larger part of the solution space U^D will exceed the remaining budget. This limits the number of branches that need to be explored, or the branches that are explored do not need to be explored to their full depth D, but only as fast as the remainder of the budget allows. This results in the time required per modification dropping near the end of the simulation run.

Results in computation time are measured using the procedure in section 3-2-3.

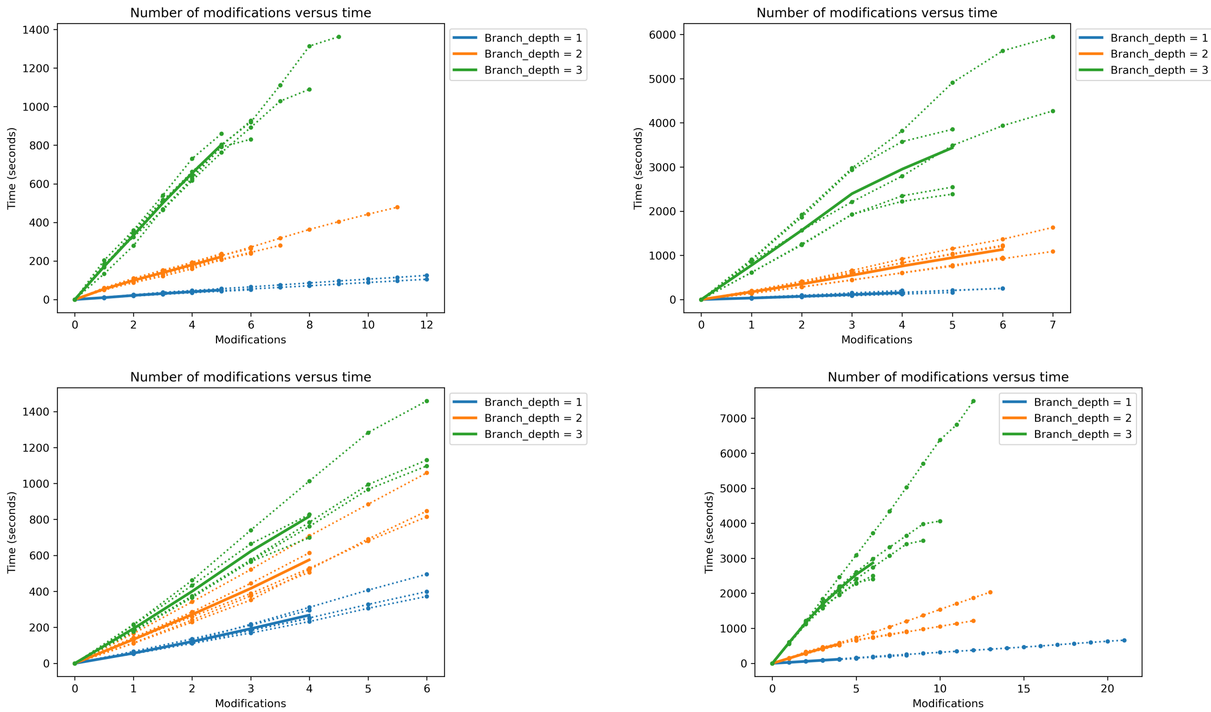

Figure 4-8: Computation time versus the number of installed modifications. The runs are ordered as: top left: Setup 1, top right: Setup 2, bottom left: Setup 3, bottom right: Setup 4. Individual runs are shown as dashed lines, rolling averages as solid lines.

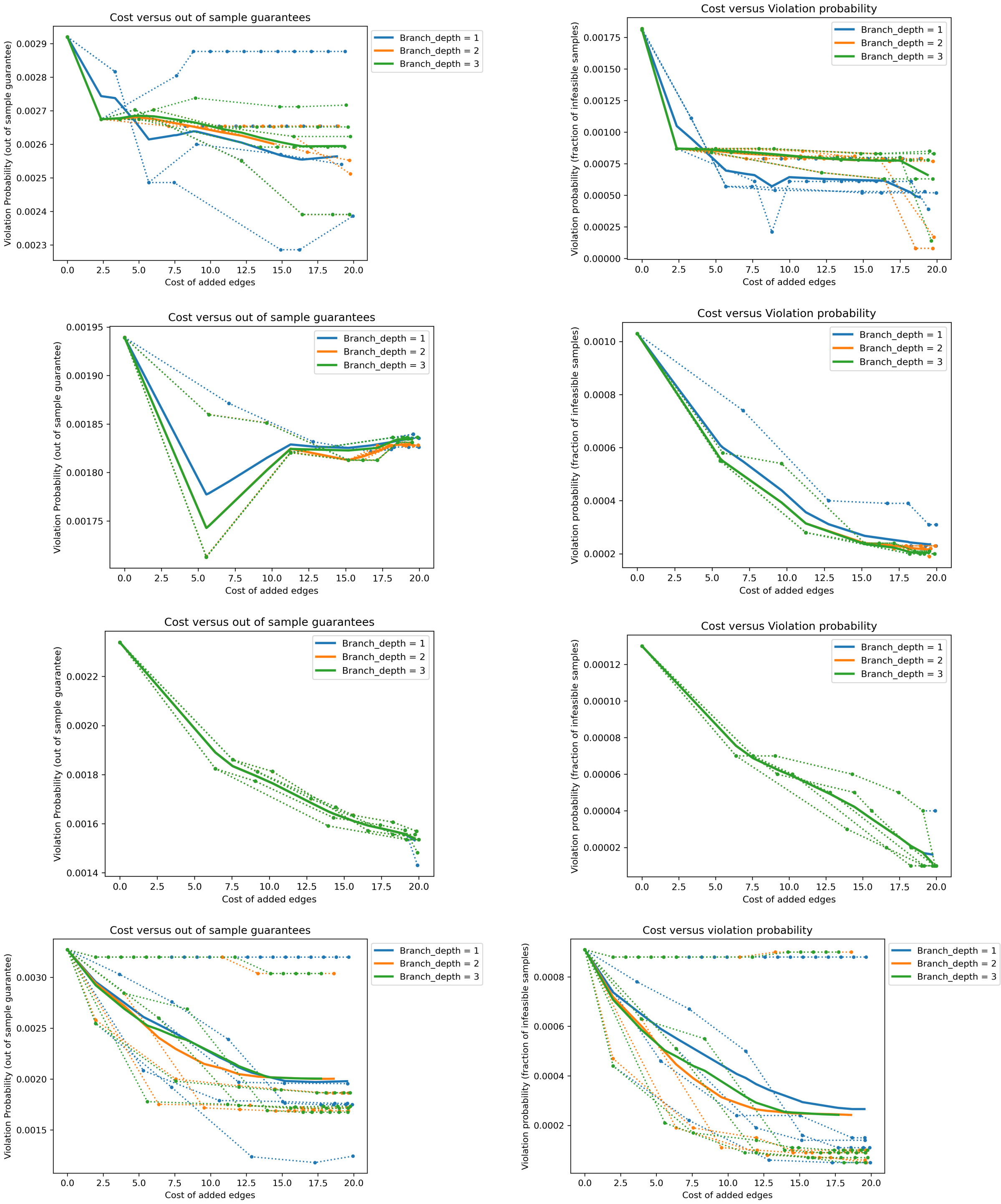

Figure 4-9: Violation probability versus modification cost both using the scenario approach (left) as well as the Monte-Carlo approach (right). Each row corresponds to an initial grid setting. Individual runs are shown as dashed lines, rolling averages as solid lines.

4-4 Analysis of results

When analyzing the modifications made, we can plot the relative improvement of each modification, both in terms of Monte-Carlo performance as well as scenario performance. These values correspond with the (negated) values in the second row of 3-16 and 3-14, respectively.

$$\text{Improvement}_{MC}(u) = \frac{\hat{f}_{\mathcal{G}}^{MC} - \hat{f}_{\mathcal{G}+u}^{MC}}{\hat{f}_{\mathcal{G}}^{MC}}$$
$$\text{Improvement}_{Sc}(u) = \frac{\hat{f}_{\mathcal{G}}^{Sc} - \hat{f}_{\mathcal{G}+u}^{Sc}}{\hat{f}_{\mathcal{G}}^{Sc}}$$

We can plot these values together for all runs comparing the Monte-Carlo approach to grid expansion and the scenario approach to grid expansion. This is shown in figure 4-10.

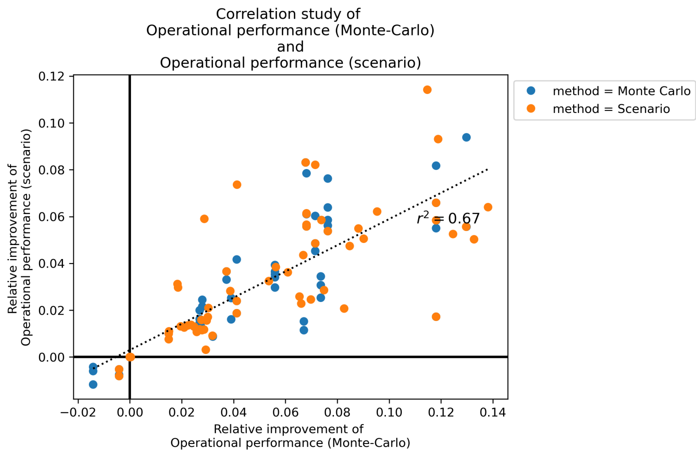

Figure 4-10: Relative improvement of Monte-Carlo operational performance versus relative improvement of scenario operational performance

We can reasonably conclude that a 'good' modification selected using a scenario approach (one that will yield an operational improvement) will also result in a comparable improvement when testing operational performance using the Monte-Carlo approach. This conclusion can be used to quickly filter a large modification set U for the most promising modifications, and make an informed selection of this set when applying the Monte-Carlo approach is preferred.

For the other parameter studies, similar results as figure 4-10 were produced. The results in the lower right or upper left quadrant (an improvement in one operational performance metric, but a deterioration in the other) were few and all close to the origin. This behavior can be stopped by using a more strict stopping criterion, specifically c_{mod}.

Chapter 5

Conclusion

Electricity grids around the world are struggling with grid congestion. The amount of power demanded or the power supplied do not match or overload the local power lines. This grid congestion has been exacerbated by the increased supply of intermittent power sources. To alleviate this issue, a lot of research has been done in expanding the power grid to cope with the peaks of power loads, ensuring steady delivery to consumers.

The objective of grid expansion planning is to modify a grid such that the operational efficiency of the grid improves, and the grid as a whole becomes more reliable. This can be a challenging problem as a balance must be struck between this improvement, and the capital expenditures associated with modifying the grid. Currently, this balance is often struck using a Monte-Carlo simulation of the power grid.

Some power grid operation studies apply scenario optimization to the operation of power grids. Scenario optimization optimizes using a single sample set, guaranteeing a reliability claim for all new samples with some confidence. This allows for robust optimization of the grid, ensuring that blackouts are guaranteed to be rare and the grid is used efficiently, given the uncertainty in power loads.

The goal of this thesis was to develop a method of applying scenario optimization to grid expansion, exploiting the robust nature of this optimization technique. The main research question was to find if the scenario approach held up with the Monte-Carlo approach. The second research question was if using the information given on reliability by scenario optimization explicitly in the expansion optimization yielded better results. The last research aim and to find any drawbacks or advantages of the scenario approach that can be used to achieve even better results.

To this end, a power flow model was chosen, and three optimization models were developed; Using the Monte-Carlo approach, the scenario approach and the scenario approach for different development horizons. These models were developed in such a way that the model architecture was consistent between the three. Using these optimization models, three parameter studies were run comparing optimization models, and optimizing with or without explicitly using robustness information from the scenario optimization approach.

The main conclusion is that the scenario approach can be used in grid expansion programs. The results are comparable to or exceed the results when using a Monte-Carlo approach.

The second conclusion is that modifications associated with a sufficient improvement in one performance metric, either using Monte-Carlo optimization or scenario optimization, are associated with an improvement in the other. This allows a grid designer that requires some result using the Monte-Carlo approach to first find some selection of most promising results using the scenario approach, only to test those using the Monte-Carlo approach.

This is only beneficial because of the large computational efficiency improvement with the scenario approach over the Monte-Carlo approach, which is the third conclusion of this thesis. Changing the optimization method from Monte-Carlo to scenario has yielded computation time improvement ranging from a factor 20 to a factor 26.

the fourth conclusion is that optimizing whilst explicitly using the information on support constraints has not yielded improved results in the short term, to eventually converge with the optimization results not using that information. There is no proof that optimization over the number of support constraint had any tangible depressive effect on this number in the long run.

The fifth and final conclusion is that expanding the development horizon of the scenario approach could effectively convert the large computational advantage over the Monte-Carlo approach into improved results for some graphs. However, there are some diminishing returns when expanding the horizon, where the added result improvement does not hold up against the large computational penalty.

Whilst the results over all initial power grids were mostly consistent, there is still some selection bias present. For example, for a grid to even be suitable for grid expansion using the scenario approach, it has to have some level of reliability, as a feasible sample set is required for the scenario approach. The Monte-Carlo method does not have this requirement. By only optimizing grids that were in some sense already robust to a certain extend, we select a subset of all possible grids for this research. This limits the area of application of the conclusions presented in this thesis.

The scenario approach optimization results and guarantees hold distribution-free. However, this is not necessarily the case for the Monte-Carlo approach. As a result of this, the probability density functions the grids in this study used might influence the result of the latter, but not the former. This could possibly have skewed results in favor of one over the other, but during the research process no results were found that contradict the results presented in this report.

Chapter 6

Discussion

This chapter outlines the points for discussion that have arisen from the results presented in this thesis. Some of these discussion points can be considered recommendations for topics of further research.

The results in this thesis may be biased or not representative for a grid expansion of a different form. Results may not be applicable to a graph of different form or reliability level, or optimization settings. Further research could be conducted to find the boundaries of the applicability of the conclusions of this thesis.

This thesis only considered the possibility of adding edges to a power grid. Further research could focus to expanding that decision space to other types of grid modification, e.g. the addition of power sources, power drains, demand-response capacity or even buffers such as battery energy storage systems.

For a larger graph, the feasible sample set consists of more samples, and therefore the feasible subdomain is more saturated and smaller. This might result in the midpoint being more representative of the entire feasible subdomain than for smaller graphs. If this is the case, the performance comparison of the Monte-Carlo approach to grid expansion and the scenario approach to grid expansion skewes more in favor of the Monte-Carlo approach for larger grids. Further studies could be conducted in finding the effect of graph size on the relative performance of Monte-Carlo expansion planning and scenario expansion planning.

The results and conclusions of this study might be biased by the chosen power flow model. By running the same expansion program, using a different power flow optimization model, a further research study might find that the conclusions are only applicable for this chosen power flow model, or applicable elsewhere too.

The aim of this study has been to find the best improvement from a set of possible modifications. There is however no guarantee that the path chosen is globally optimal. Further research could be aimed at describing the probability that the result of a greedy approach to grid expansion is also part of the modification sequence leading to the global optimum. This research could also explore the influence depth D and breadth B might have on this probability.

We have shown that it is possible to find performance improvements using scenario optimization. However, we did not find an accurate predictor on how large that improvement will be. Further research could aim at estimating the possible performance gain, given some initial grid settings. This would be informative on deciding on what power grids to improve when a total budget between multiple grids is limited.

The assertion of scenario optimization and its application to optimal power grid operation holds distribution-free. However, this guarantee is not proven for the scenario approach to grid expansion. Further research could be aimed at proving this, and finding if the same holds for the Monte-Carlo approach. If the latter is not true, it could find some criteria on the distributions that predict the performance comparison of the Monte-Carlo approach to grid expansion and the scenario approach to grid expansion.

The optimization method assumes full information on the grid and the probability distributions. In practice, this information might not be fully known, or changing over time, e.g. the power loads of an expanding neighbourhood both change over time and are not exctly known at each moment. Further studies could find the effect of this model uncertainty on the grid expansion performance programs, and possible mitigation avenues.

The final recommendation concerns the comparison of the scenario approach to other optimization models. While we did choose to compare the novel scenario approach to grid expansion with the prevalent greedy Monte-Carlo approach, this Monte-Carlo approach is not the only one applied to grid expansion. Further research could test if the scenario approach also holds up to these other optimization approaches. This further research could also test the conclusion that a performance improvement in scenario optimization operational performance also signals an improvement in the other optimization method's operational performance.

Appendix

6-1 Arguments and derivations

6-1-1 Beta distribution of violation probability

The argumentation in [34] is as follows:

- Consider a fully supported problem of k support constraints
- We know, for this fully supported problem, that the violation probability follows a beta-distribution of $beta(k, N-k+1)$
- We can embed this problem into a larger problem with $n_\theta > k$ unconstrained control parameters, independent of the first k control parameters, and still claim this same probability density curve.

  "To put the above discussion on solid grounds, consider a fully-supported problem in dimension k. For such a problem, the number of support constraints is k with probability 1. It is not hard to embed this problem into another problem that has d optimization variables while it continues to have k support constraints with probability 1, so that $s^*_N = k$ *with probability 1."*
- The requirement that this holds distribution-free only marginally increases risk.

  "The interpretation is that the number of support constraints carries the fundamental information to judge the risk, and the residual uncertainty in the risk after the number of support constraints has been seen (two samples of scenarios that lead to the same number of support constraints may carry a different risk) is only marginally increased by requiring that the result holds distribution-free."

We formulated our optimization constraint as two linear constraints, namely

$$
\begin{aligned}
S^*_{min} \leq IS^* \leq S^*_{max} \\
b_{min} \leq Z^* S^* \leq b_{max}
\end{aligned}
$$

Where we can have at most n_θ support constraints. Given that both I, Z_P and Z_Q are invertible by definition, we know they are all full rank.

This means that for all complexity levels $k \leq n_\theta$ we can construct such a basis transformation that translates the optimization problem into a problem with k constrained parameters, and $n_\theta - k$ unconstrained parameters.

We can then decompose the problem into an embedment of a fully supported problem of control dimension k and the larger optimization of control dimension $n_\theta - k$, giving us the option to exploit the knowledge on violation probability probability density curves of fully supported problems.

Lastly, the statement on this result being distribution-free furhter motivates the assumption that f_{V_1} and f_{V_2} in equation 3-1 describe independent variables.

6-1-2 Arguments on computational complexity

6-1-2-1 Branching without removing duplicates

We are showing that for a branching program of depth D and breadth B and n nodes. The number of required optimal power flow computations, when not removing duplicate branches, is equal to

$$\mathsf{N}^B_{OPF,dup} = \sum_{h=0}^{D-1} \left(\frac{n(n-1)}{2} \cdot B^h \right)$$

To show this, we start out with the starting position, with only one branch. To investigate all candidate modifications, we have to explore the entirety of U, which is

$$|U| = \frac{n(n-1)}{2}$$

Then, the first level of branches are chosen from this list of modifications, and evaluated, with again the entirety of U. The total number of tested modifications is now the sum of the modifications checked on the first level and the number of modifications checked in the second.

$$\frac{n(n-1)}{2} + \frac{n(n-1)}{2} B$$

From each of those branches, B new branches appear, leading to a total $B \cdot B$ new sets of $\frac{n(n-1)}{2}$ modifications needed to be checked. The total number of modifications is now

$$\frac{n(n-1)}{2} + \frac{n(n-1)}{2} B + \frac{n(n-1)}{2} B^2$$

We ccan continue this until $D-1$, where only the leaf nodes are explored and no new branches are made. This results in the original sum

$$\mathsf{N}^B_{OPF,dup} = \sum_{h=0}^{D-1} \left(\frac{n(n-1)}{2} \cdot B^h \right)$$

6-1-2-2 Branching with removing duplicates

We are showing that for a branching program of depth D and breadth B and n nodes. The number of required optimal power flow computations is at least

$$\mathsf{N}_{OPF}^{B} \geq \sum_{h=0}^{D-1} \left(\frac{n(n-1)}{2} \cdot \frac{(B+h-1)!}{h!(B-1)!} - \left(\frac{(B+h-1)!}{h!(B-1)!} B - \frac{(B+h)!}{(h+1)!(B-1)!} \right) \right)$$

This derivation is based on the main point that at each step into the development horizon, we assume the least possible number of branches at depth h and breadth B.

To find the minimum number of branches at depth h and breadth B, we assume that all branch paths only take improvements from a subset U_B of U with $|U_B| = B$. Now, the number of unique branches in the set of development paths is set by the combination of h draws from these B improvements in set U_B, allowing for repetition. Calculating all modifications for each those branches results in

$$\frac{n(n-1)}{2} \cdot \frac{(B+h-1)!}{h!(B-1)!}$$

From this minimal set, we calculate the improvement for all $\frac{n(n-1)}{2}$ modifications per branch. of those child modifications, we want to discard the duplicate graphs as a result of the similarities the parent branches have. The number of duplicate child modifications is given by the number of child branches created using $B \in U_B$ from those parent branches, minus the number of unique branches, given by the combination of $h+1$ draws from these B improvements in set U_B, allowing for repetition.

$$\frac{(B+h-1)!}{h!(B-1)!} B - \frac{(B+h)!}{(h+1)!(B-1)!}$$

Repeating this along all levels until $D-1$, where only the leaf modifications are checked and no new branches created, we find

$$\mathsf{N}_{OPF}^{B} \geq \sum_{h=0}^{D-1} \left(\frac{n(n-1)}{2} \cdot \frac{(B+h-1)!}{h!(B-1)!} - \left(\frac{(B+h-1)!}{h!(B-1)!} B - \frac{(B+h)!}{(h+1)!(B-1)!} \right) \right)$$

6-2 Initial grids

Below are all initial graphs and their specifications. For each grid, all information is shown for each node in terms of type, location, S_{min} and S_{max}, V_{min} and V_{max} and the sampling probability density function. All grids are scaled such that the maximum distance covered by any possible edge is 1, or $\max_{u \in U} d(u) = 1$. Edges and their admittance are also given.

Initial grid 1

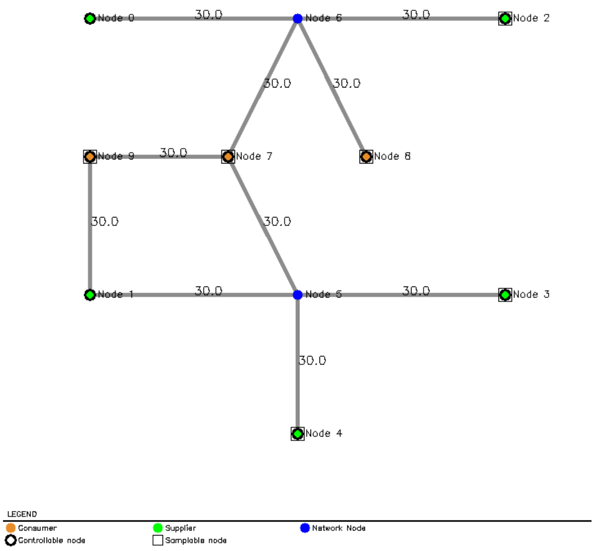

Figure 6-1: Initial grid 1

Node	Type	Location	Control authority	Voltage constraints	Sampling density
0	Supplier	$(0, 0)$	$[0, 0.2]$	$[0.95, 1.05]$	(-)
1	Supplier	$(0, 2)$	$[0, 0.2]$	$[0.95, 1.05]$	(-)
2	Supplier	$(3, 0)$	$[-0.005, 0.005]$	$[0.95, 1.05]$	$\mathcal{N}(0.1, 0.03^2)$
3	Supplier	$(3, 2)$	$[-0.005, 0.005]$	$[0.95, 1.05]$	$\mathcal{N}(0.1, 0.03^2)$
4	Supplier	$(1.5, 3)$	$[-0.005, 0.005]$	$[0.95, 1.05]$	$\mathcal{N}(0.1, 0.03^2)$
5	Network	$(1.5, 2)$	(-)	$[0.95, 1.05]$	(-)
6	Network	$(1.5, 0)$	(-)	$[0.95, 1.05]$	(-)
7	Consumer	$(1, 1)$	$[-0.0125, 0.0125]$	$[0.95, 1.05]$	$\mathcal{N}(-0.25, 0.075^2)$
8	Consumer	$(2, 1)$	$[-0.0125, 0.0125]$	$[0.95, 1.05]$	$\mathcal{N}(-0.25, 0.075^2)$
9	Consumer	$(0, 1)$	$[-0.0125, 0.0125]$	$[0.95, 1.05]$	$\mathcal{N}(-0.25, 0.075^2)$

Table 6-1: Grid node positions for setup 1. Node positions are scaled such that the largest (possible) edge is 1 unit length.

Node 1	Node 2	Admittance
Slack	0	30
0	6	30
2	6	30
6	8	30
6	7	30
7	9	30

Node 1	Node 2	Admittance
1	9	30
1	5	30
5	7	30
3	5	30
4	5	30

Table 6-2: Lines of setup 1

Initial grid 2

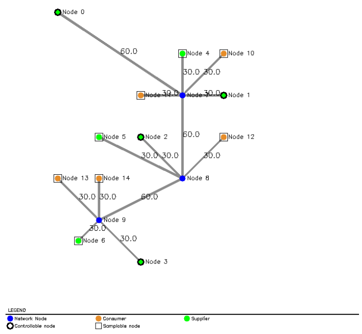

Figure 6-2: Initial grid 2

Node	Type	Location	Control authority	Voltage constraints	Sampling density
0	Supplier	$(0, 0)$	$[0, 0.25]$	$[0.95, 1.05]$	(-)
1	Supplier	$(2, 1)$	$[0, 0.25]$	$[0.95, 1.05]$	(-)
2	Supplier	$(1, 1.5)$	$[0, 0.25]$	$[0.95, 1.05]$	(-)
3	Supplier	$(1, 3)$	$[0, 0.25]$	$[0.95, 1.05]$	(-)
4	Supplier	$(1.5, 0.5)$	$[-0.005, 0.005]$	$[0.95, 1.05]$	$\mathcal{N}(0.1, 0.03^2)$
5	Supplier	$(0.5, 1.5)$	$[-0.005, 0.005]$	$[0.95, 1.05]$	$\mathcal{N}(0.1, 0.03^2)$
6	Supplier	$(0.25, 2.75)$	$[-0.005, 0.005]$	$[0.95, 1.05]$	$\mathcal{N}(0.1, 0.03^2)$
7	Network	$(1.5, 1)$	(-)	$[0.95, 1.05]$	(-)
8	Network	$(1.5, 2)$	(-)	$[0.95, 1.05]$	(-)
9	Network	$(0.5, 2.5)$	(-)	$[0.95, 1.05]$	(-)
10	Consumer	$(2, 0.5)$	$[-0.0125, 0.0125]$	$[0.95, 1.05]$	$\mathcal{N}(-0.25, 0.075^2)$
11	Consumer	$(1, 1)$	$[-0.0125, 0.0125]$	$[0.95, 1.05]$	$\mathcal{N}(-0.25, 0.075^2)$
12	Consumer	$(2, 1.5)$	$[-0.0125, 0.0125]$	$[0.95, 1.05]$	$\mathcal{N}(-0.25, 0.075^2)$
13	Consumer	$(0, 2)$	$[-0.0125, 0.0125]$	$[0.95, 1.05]$	$\mathcal{N}(-0.25, 0.075^2)$
14	Consumer	$(0.5, 2)$	$[-0.0125, 0.0125]$	$[0.95, 1.05]$	$\mathcal{N}(-0.25, 0.075^2)$

Table 6-3: Grid node positions for setup 2. Node positions are scaled such that the largest (possible) edge is 1 unit length.

Node 1	Node 2	Admittance
Slack	0	30
0	7	60
7	8	60
8	9	60
1	7	30
4	7	30
7	10	30
7	11	30

Node 1	Node 2	Admittance
2	8	30
5	8	30
8	12	30
3	9	30
9	13	30
9	14	30
6	9	30

Table 6-4: Lines of setup 2

Initial grid 3

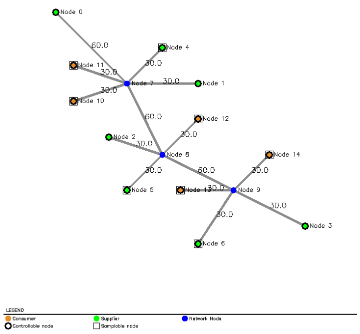

Figure 6-3: Initial grid 3

Node	Type	Location	Control authority	Voltage constraints	Sampling density
0	Supplier	$(0, 0)$	$[0, 0.25]$	$[0.95, 1.05]$	(-)
1	Supplier	$(2, 1)$	$[0, 0.25]$	$[0.95, 1.05]$	(-)
2	Supplier	$(0.75, 1.75)$	$[0, 0.25]$	$[0.95, 1.05]$	(-)
3	Supplier	$(3.5, 3)$	$[0, 0.25]$	$[0.95, 1.05]$	(-)
4	Supplier	$(1.5, 0.5)$	$[-0.0075, 0.0075]$	$[0.95, 1.05]$	$\mathcal{N}(0.15, 0.045^2)$
5	Supplier	$(1, 2.5)$	$[-0.0075, 0.0075]$	$[0.95, 1.05]$	$\mathcal{N}(0.15, 0.045^2)$
6	Supplier	$(2, 3.25)$	$[-0.0075, 0.0075]$	$[0.95, 1.05]$	$\mathcal{N}(0.15, 0.045^2)$
7	Network	$(1, 1)$	(-)	$[0.95, 1.05]$	(-)
8	Network	$(1.5, 2)$	(-)	$[0.95, 1.05]$	(-)
9	Network	$(2.5, 2.5)$	(-)	$[0.95, 1.05]$	(-)
10	Consumer	$(0.25, 1.25)$	$[-0.0125, 0.0125]$	$[0.95, 1.05]$	$\mathcal{N}(-0.25, 0.075^2)$
11	Consumer	$(0.25, 0.75)$	$[-0.0125, 0.0125]$	$[0.95, 1.05]$	$\mathcal{N}(-0.25, 0.075^2)$
12	Consumer	$(2, 1.5)$	$[-0.0125, 0.0125]$	$[0.95, 1.05]$	$\mathcal{N}(-0.25, 0.075^2)$
13	Consumer	$(1.75, 2.5)$	$[-0.0125, 0.0125]$	$[0.95, 1.05]$	$\mathcal{N}(-0.25, 0.075^2)$
14	Consumer	$(3, 2)$	$[-0.0125, 0.0125]$	$[0.95, 1.05]$	$\mathcal{N}(-0.25, 0.075^2)$

Table 6-5: Grid node positions for setup 3. Node positions are scaled such that the largest (possible) edge is 1 unit length.

Node 1	Node 2	Admittance
Slack	0	30
0	7	60
7	8	60
8	9	60
1	7	30
4	7	30
7	10	30
7	11	30

Node 1	Node 2	Admittance
2	8	30
5	8	30
8	12	30
3	9	30
9	13	30
9	14	30
6	9	30

Table 6-6: Lines of setup 3

Initial grid 4

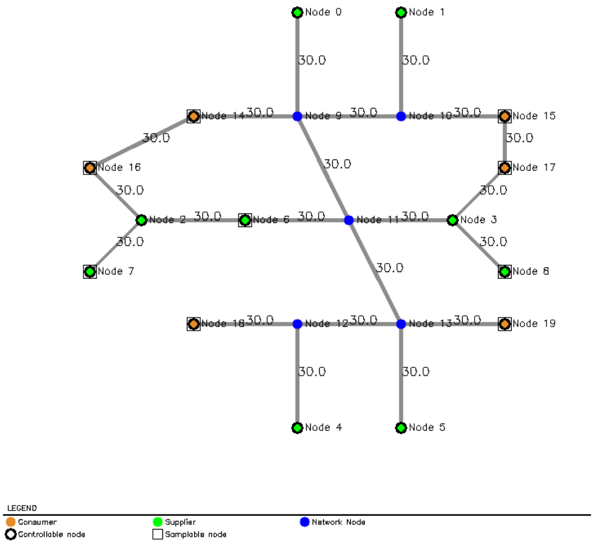

Figure 6-4: Initial grid 4

Node	Type	Location	Control authority	Voltage constraints	Sampling density
0	Supplier	$(2, 0)$	$[0, 0.25]$	$[0.95, 1.05]$	(-)
1	Supplier	$(3, 0)$	$[0, 0.25]$	$[0.95, 1.05]$	(-)
2	Supplier	$(0.5, 2)$	$[0, 0.25]$	$[0.95, 1.05]$	(-)
3	Supplier	$(3.5, 2)$	$[0, 0.25]$	$[0.95, 1.05]$	(-)
4	Supplier	$(2, 4)$	$[0, 0.25]$	$[0.95, 1.05]$	(-)
5	Supplier	$(3, 4)$	$[0, 0.25]$	$[0.95, 1.05]$	(-)
6	Supplier	$(1.5, 2)$	$[-0.0075, 0.0075]$	$[0.95, 1.05]$	$\mathcal{N}(0.15, 0.03^2)$
7	Supplier	$(0, 2.5)$	$[-0.0075, 0.0075]$	$[0.95, 1.05]$	$\mathcal{N}(0.15, 0.03^2)$
8	Supplier	$(4, 2.5)$	$[-0.0075, 0.0075]$	$[0.95, 1.05]$	$\mathcal{N}(0.15, 0.03^2)$
9	Network	$(2, 1)$	(-)	$[0.95, 1.05]$	(-)
10	Network	$(3, 1)$	(-)	$[0.95, 1.05]$	(-)
11	Network	$(2.5, 2)$	(-)	$[0.95, 1.05]$	(-)
12	Network	$(2, 3)$	(-)	$[0.95, 1.05]$	(-)
13	Network	$(3, 3)$	(-)	$[0.95, 1.05]$	(-)
14	Consumer	$(1, 1)$	$[-0.0125, 0.0125]$	$[0.95, 1.05]$	$\mathcal{N}(-0.25, 0.075^2)$
15	Consumer	$(4, 1)$	$[-0.0125, 0.0125]$	$[0.95, 1.05]$	$\mathcal{N}(-0.25, 0.075^2)$
16	Consumer	$(0, 1.5)$	$[-0.0125, 0.0125]$	$[0.95, 1.05]$	$\mathcal{N}(-0.25, 0.075^2)$
17	Consumer	$(4, 1.5)$	$[-0.0125, 0.0125]$	$[0.95, 1.05]$	$\mathcal{N}(-0.25, 0.075^2)$
18	Consumer	$(1, 3)$	$[-0.0125, 0.0125]$	$[0.95, 1.05]$	$\mathcal{N}(-0.25, 0.075^2)$
19	Consumer	$(4, 3)$	$[-0.0125, 0.0125]$	$[0.95, 1.05]$	$\mathcal{N}(-0.25, 0.075^2)$

Table 6-7: Grid node positions for setup 4. Node positions are scaled such that the largest (possible) edge is 1 unit length.

Node 1	Node 2	Admittance
Slack	0	30
Slack	1	30
0	9	30
1	10	30
9	10	30
10	15	30
9	14	30
14	16	30
2	16	30
2	6	30
9	11	30
6	11	30

Node 1	Node 2	Admittance
3	11	30
3	17	30
3	8	30
2	7	30
12	13	30
12	18	30
4	12	30
11	13	30
5	13	30
13	19	30
15	17	30

Table 6-8: Lines of setup 4

6-3 Code

The codebase is split up into 7 blocks, with each block their own functionalities.

GraphClass	Basic class definition script defining Graphs, Nodes and Edges and the functionality to manipulate these
CalcTools	Toolbox for standard calculations such as a-posteriori ϵ or N_{log}
GraphOPF	Optimal power flow calculations
GraphOptimizationLoop	Optimization loop
GraphValidation	Result validation stage
GraphParameterStudy	Main script managing an entire parameter study
GraphPlot	Plotting a graph for intermediate results

Results are stored (if this is enabled) in a folder made with the current timestamp in the 'Runs' folder.

The code is shown below. There are some functionalities which have been developed (e.g. Optimal Power Flow for multiple time instances with corresponding density functions and clustering the graph to simplyify the expansion problem) but not presented in this report. Code corresponding to these functionalities is marked with the comment "[Not used for final report]".

At the end of the codebase are some examples of function calls to run parameter or correlation studies, and the initialization of initial graph 1.

GraphClass

```
import numpy as np
from scipy import optimize as op
import copy

import CalcTools

class MergeError(Exception): #[Not used for final report]
    pass

class Distribution:
    """
    A class used to represent a probability distribution to sample from

    ...

    Attributes
    ----------
    function: function
        A function to draw samples from
    *args:
        Arguments to be passed into function
    ----------

    ...

    """
    def __init__(self,function,*args,multiplier = 1, name = ""):
        self.function = function
        self.args = args

        self.multiplier = multiplier

        self.name = name

    def Sample(self,*args):
        """Returns a sample according to the distribution function"""
        return self.multiplier*self.function(*self.args,*args)

```

```python
    def MultiSample(self,N,M=1):
        """
        Returns an array of N samples according to the distribution function

        N: int
            Number of samples
        M: int [Not used for final report]
            Number of time instances
        """
        return np.array([[self.Sample(int(i*(24/M)+12)) for i in range(M)] for j in range(N)])

    def __str__(self):
        """Returns a string representing self"""
        if self.name == "":
            return str(self.function).split(".")[2]+str(self.args)+", "+str(self.multiplier)
        else:
            return self.name+": "+str(self.args)+", "+str(self.multiplier)


class Node:
    """
    A class used to represent a probability distribution to sample from

    ...

    Attributes
    ----------
    controllable: bool
        Determines if node can dispatch a power load on demand
    color:        3x1 array
        Array defining BGR color
    TypeName:     string
        Legible type name designation
    connections:  set
        A set of connected Edges
    distribution: Distribution
        A distribution to sample from, None if no sampling at this node
    constraints:  dictionary
        Voltage level constraints at this node (low/high)
        Power level constraints at this node (ctrl_low/ctrl_high)
    name:         string
        Name of node
    position:     tuple
        Position of node
    Clustered:    bool [Not used for final report]
        Used in clustering procedure (standard: False)
    ----------

    ...

    """
    controllable = False
    samplable = False
    color = np.array([0,0,0],dtype=float)
    TypeName = "Standard Node type"

    def __init__(self, distribution = None, constraints = dict(),name = "No name",position = (None,
     None),Clustered=False):
        self.connections = set()

        self.distribution = distribution

        self.constraints = {self:constraints}
        Standard = {"low":0.95,"high":1.05,"ctrl_low":-0.05,"ctrl_high":0.05}
        for key in Standard.keys():
            if key not in self.constraints[self]:
                if key in {"ctrl_low","ctrl_high"}:
                    num_samp = 1000
                    samp = self.MultiSample(num_samp,24)
                    self.constraints[self][key] = Standard[key].real*np.abs(np.average(samp)).real
                else:
                    self.constraints[self][key] = Standard[key]

        self.Update_tags()

        self.name = str(name)
        self.position = position

        self.Clustered = Clustered

        self.Check_child_class()
        return

    def Update_tags(self):
        """
        Update the controllable and samplable tags
        """
        self.controllable = not(self.constraints[self]["ctrl_low"] == self.constraints[self]["
            ctrl_high"])
        self.samplable = False if self.distribution is None else True
```

```
        return 1

    def Connect(self,Connection):
        """
        Adds an edge to connections

        Connection: Edge
            Edge object to connect to

        """
        self.connections.add(Connection)
        return

    def Disconnect(self,Connection):
        """
        Disconnect from edge

        Connection: Edge
            Edge object to disconnect from

        """
        if Connection in self.connections:
            self.connections.discard(Connection)
        return

    def Disconnect_all(self):
        """
        Disconnect from all edges connected to this node
        """
        for conn in self.connections:
            self.Disconnect(conn)
        return

    def Merge(self,other,Admittance_multiplier):
        """
        [Not used for final report]
        Merge self onto other, with admittance multiplier

        other: Node
            Node to merged on to
        Admittance_multiplier: float
            Part of power of self mapped onto other (projection of loads of self)
        """
        if self.Clustered:
            raise MergeError("Self already merged node")

        constraints_new = self.constraints[self]

        for key in other.constraints[other].keys():
            if key in constraints_new.keys():
                if key == "high":
                    constraints_new[key] = min(constraints_new[key],other.constraints[other][key])
                elif key == "low":
                    constraints_new[key] = max(constraints_new[key],other.constraints[other][key])
                elif key == "ctrl_high":
                    constraints_new[key] = constraints_new[key]+other.constraints[other][key]/Admittance_multiplier
                elif key == "ctrl_low":
                    constraints_new[key] = constraints_new[key]+other.constraints[other][key]/Admittance_multiplier
                else:
                    raise NotImplementedError("Constraint type not implemented in merge algorithm")
            else:
                constraints_new[key] = other.constraints[other][key]

        name1 = [self.name] if self.name[:6] != "Merged" else self.name[21:].split(", ")
        name2 = [other.name] if other.name[:6] != "Merged" else other.name[21:].split(", ")
        name_new = "Merged node of nodes "+", ".join([*name1,*name2])


        if self.distribution is None:
            pass
        elif isinstance(self.distribution,list):
            for dist in self.distribution:
                dist.multiplier /= Admittance_multiplier
        else:
            self.distribution.multiplier /= Admittance_multiplier


        dist1 = [] if self.distribution == None else self.distribution if isinstance(self.distribution,list) else [self.distribution]
        dist2 = [] if other.distribution == None else other.distribution if isinstance(other.distribution,list) else [other.distribution]
        dist_new = [*dist1,*dist2]


        if dist_new == []:
            dist_new = None
```

```
        Merged_node = Node(distribution = dist_new, \
                           constraints = constraints_new, \
                           name = name_new, \
                           position = other.position)

        Merged_node.Check_child_class()

        Edges = []

        while len(other.connections)>0:
            edge,*_ = other.connections
            for node in edge.connections:
                if node == self or node == other:
                    pass
                else:
                    Edges += [Edge(Merged_node,node,edge.admittance,edge.constraints[edge])]
            edge.Disconnect()

        Merged_node.Clustered = True
        return Merged_node

    def Check_child_class(self):
        """
        Assigns self into the appropriate child class.
        This has no effect on performance, only on TypeName and Color tags used by GraphPlot.py
        """
        signlist_samp = [0]

        if self.samplable:
            if isinstance(self.distribution,list):
                for dist in self.distribution:
                    signlist_samp += [dist.multiplier]
            else:
                signlist_samp += [self.distribution.multiplier]

            if max(max(signlist_samp),-min(signlist_samp)) <= 10**-6:
                signlist_samp = [0]

        signlist_cont = [0]
        if self.controllable:
            #Only include controllability if it is comparable to the expected value of the sampled load
            num_samp = 100
            samp = self.MultiSample(num_samp,24)
            factor = 2

            if abs(self.constraints[self]["ctrl_low"]*factor) >= np.abs(np.average(samp)):
                signlist_cont += [self.constraints[self]["ctrl_low"].real]

            if abs(self.constraints[self]["ctrl_high"]*factor) >= np.abs(np.average(samp)):
                signlist_cont += [self.constraints[self]["ctrl_high"].real]

            if max(max(signlist_cont),-min(signlist_cont)) <= 10**-6:
                signlist_cont = [0]

        signlist = [*signlist_samp,*signlist_cont]

        if not(self.samplable or self.controllable):
            #Network
            self.__class__ = Network_node
            self.distribution = None
        else:
            if (np.array(signlist)>=0).all():
                #Supplier
                self.__class__ = Supplier
            elif (np.array(signlist)<=0).all():
                #Consumer
                self.__class__ = Consumer
            else:
                #Prosumer
                self.__class__ = Prosumer

        return self.__class__


    def Sample(self):
        """
        Draw a sample of own power load probability distribution
        """
        return self.dist.Sample()

    def MultiSample(self,N,M=1):
        if isinstance(self.distribution,Distribution):
            Samples = self.distribution.MultiSample(N,M)
        elif isinstance(self.distribution,list):
            Samples = np.sum(np.concatenate(([dist.MultiSample(N,M) for dist in self.distribution]),axis=0),axis=0)
        elif self.distribution is None:
            Samples = np.zeros((N,M))
        else:
```

```
            raise TypeError("Distribution attribute of unexpected type: "+str(type(self.distribution)))
        return Samples


    def __str__(self):
        """
        Returns a string representing self
        """
        return self.TypeName + " " + str(self.name) + " at "+str(self.position)

    def Summarize(self):
        """
        Returns a string representing self and children
        """
        Summary = str(self)+"; Constraints: "+str(self.constraints)+ "; Distribution: "+str(self.distribution)
        return Summary






class Supplier(Node):
    """Parent class for nodes that nominally supply power"""
    TypeName = "Supplier"
    color = np.array([0,255,0],dtype=float)
    pass

class Consumer(Node):
    """Parent class for nodes that nominally consume power"""
    TypeName = "Consumer"
    color = np.array([40, 140, 230],dtype=float)
    pass

class Prosumer(Node):
    """Class for nodes that both supply and consume power and are uncontrollable (such as a home with solar roof), child of Supplier_uncontrollable and Consumer_uncontrollable types"""
    TypeName = "Prosumer (uncontrollable)"
    color = np.array([30, 250, 250],dtype=float)
    pass

class Network_node(Node):
    """Class for nodes with no power draw or supply, only ment for distribution and transport of power, child of Node (not considered uncontrollable since not stochastic by nature)"""
    color = np.array([255,0,0],dtype=float)
    TypeName = "Network Node"
    pass



class Edge:
    """
    A class used to represent an edge (power cable) on the grid

    ...

    Attributes
    ----------
    color:        3x1 array
        Array defining BGR color
    connections:  set
        A set of connected Nodes
    admittance: complex
        The complex admittance of this edge
    constraints:  dictionary
        Voltage delta constraints on this edge (low/high)
    ----------

    ...

    """
    color = np.array([140,140,140],dtype=float)
    def __init__(self,Connection_in,Connection_out,admittance=None,constraints=None):
        self.connections = set()

        self.admittance = admittance

        if not(constraints == None):
            self.constraints = {self:constraints}
        else:
            standard = {"ctrl_low":1,"ctrl_high":1}
            self.constraints = {self:standard}


        self.Connect(Connection_in,Connection_out)
        return


```

```
    def Connect(self,Connection_in,Connection_out):
        """
        Connects edge to two nodes

        Connection_in:    Node
            Node one to connect to
        Connection_out:    Node
            Node two to connect to

        """
        self.connections = {Connection_in,Connection_out}
        Connection_in.Connect(self)
        Connection_out.Connect(self)

    def Disconnect(self):
        """Disconnect connected nodes from self"""
        for Node in self.connections:
            Node.Disconnect(self)
        self.connections = {}

    def __str__(self):
        """Returns a string representing self"""
        return str(list(self.connections)[0].name)+" <--> "+str(list(self.connections)[1].name)+" :
            "+str(self.admittance)



class Graph:
    """
    A class used to represent the grid, with nodes and edges contained within

    ...

    Attributes
    ----------
    Nodes_list:    list
        List of all nodes in the graph
    Edges_list:    list
        List of all edges in the graph, updated with each modification
    Index_Lookup: dictionary
        Dictionary linking nodes to their position in Nodes_list
    Theta_Lookup: dictionary
        Similar to Index_Lookup, but exclusively containing controllable nodes
    Delta_Lookup: dictionary
        Similar to Index_Lookup, but exclusively containing uncontrollable nodes
    n_nodes:       int
        Number of nodes
    n_theta:       int
        Number of controllable nodes
    n_delta:       int
        Number of uncontrollable nodes
    Conn_list:     list
        List of all node pairs connected by an edge, updated explicitly
    SBA:           float
        Slack bus admittance of the grid
    Scale:         float
        Scale of the grid
    Z_p:           Array
        Real part of impedance matrix
    Z_q:           Array
        Complex part of impedance matrix
    ----------

    ...

    """
    def __init__(self,Nodes = None,Edges = None, Slack_bus_connections = dict(), Scale = 1):
        self.Nodes_list = Nodes
        self.Edges_list = Edges
        self.Comp_Lookups()
        self.Comp_edge_sets()

        self.Slack_bus_connections = Slack_bus_connections

        self.Scale = Scale

        try:
            self.Comp_Impedance_matrices(self.Slack_bus_connections)
        except:
            pass


        return


    def __str__(self):
        """Returns a string representing self"""
        return "\n".join([str(Node) for Node in self.Nodes_list])\
            +"\n\n"\
            +"\n".join([str(Edge) for Edge in self.Edges_list])
```

```
    def Summarize(self):
        """Returns a string representing self and children"""
        Summary = "Slack bus connections: "+str(self.Slack_bus_connections)+"; Scale: "+str(self.Scale)
        Summary += "\n\n"
        Summary += "\n".join([node.Summarize() for node in self.Nodes_list])
        Summary += "\n\n"
        Summary += "\n".join([str(edge) for edge in self.Edges_list])
        return Summary

    def Add_edge(self, node_begin, node_end, *args):
        """
        Add edge to grid, connected to two nodes.
        If edge already exists, add the aspects of the "new" edge to existing one

        node_begin: Node
            First node connected to edge
        node_end:   Node
            Second node connected to edge

        *args:
            Arguments to be passed into new edge
        """
        #Check if edge already exists
        if not {self.Index_Lookup[node_begin],self.Index_Lookup[node_end]} in self.Conn_list:
            #Add edge
            edge = Edge(node_begin, node_end, *args)
            self.Edges_list += [edge]
            return edge
        else:
            return 0

    def Remove_edge(self,edge):
        """
        Removes edge from grid

        edge: Edge
            Edge to be removed
        """
         #Check if edge already exists
        if edge in self.Edges_list:
            #Add node
            self.Edges_list.remove(edge)
            edge.Disconnect()
            return 1
        else:
            return 0

    def Update_edge(self,edge,new_admittance=None,new_constraints=None):
        """
        Updates existing edge

        edge: Edge
            Edge to be upgraded

        new_admittance: complex
            Updated admittance, None if no update (default: None)
        new_constraints: complex
            Updated constraints, None if no update (default: None)
        """
        #Upgrade edge
        if new_admittance != None:
            edge.admittance = new_admittance
        if new_constraints != None:
            edge.constraints[edge] = new_constraints
        return 1

    def Import_edge_modifications(self, addition_list = None, src = None):
        """
        Import multiple edge modifications from either an array or adress and implement on self.

        addition_list: numpy array
            Numpy array with indexing
                0) begin node (index)
                1) end node (index)
                2) type of modification (1: addition, 0: upgrade)
                3) Admittance value of edge
        src: float
            Adress to be used to import addition_list from if addition_list field is empty
        """
        if addition_list == None:
            try:
                addition_list = np.loadtxt(src,dtype=float)
            except:
                raise ImportError ("Error while importing array")

        # 0,1 --> nodes
        # 3   --> admittance
        for i in range(addition_list.shape[0]):
```

```
                [begin,end,mod,admittance] = addition_list[i,[0,1,2,3]]
                print([int(begin),int(end),int(mod),admittance])
                if mod:
                    self.Add_edge(self.Nodes_list[int(begin)],self.Nodes_list[int(end)],admittance)
                else:
                    for edge in self.Edges_list:
                        begin_test,end_test = edge.connections
                        if set([self.Index_Lookup[begin_test],self.Index_Lookup[end_test]]) == set([int(
                            begin),int(end)]):
                            break
                    self.Update_edge(edge,new_admittance = admittance)

        return 1

    def Add_node(self,node,connected_nodes):
        """
        Add Node to self

        node: Node
            Node to be added
        connected_nodes: dict
            Connections of new node with complex admittances
        """
        raise NotImplementedError
        #Not implemented because project is only edge based

    def Remove_node(self,node):
        """
        Remove Node from self

        node: Node
            Node to be removed
        """
        while len(node.connections)>0 :
            edge,*_ = node.connections
            self.Remove_edge(edge)
        self.Nodes_list.remove(node)
        return 1

    def Update_nodes(self):
        """
        Passes Check_child_class function call to all nodes in graph
        """
        modified = False
        for nod in self.Nodes_list:
            temp = nod.__class__
            new_temp = nod.Check_child_class()
            if not(temp == new_temp):
                modified = True
        return modified

    def Comp_Lookups(self):
        """
        Compiles three node dictionaries to look up node-, control- and delta indices
        """
        self.Index_Lookup = dict()
        self.Theta_Lookup = dict()
        self.Delta_Lookup = dict()
        for i,Node in enumerate(self.Nodes_list):
            self.Index_Lookup[Node] = i
            if Node.controllable:
                self.Theta_Lookup[Node] = i
            if Node.samplable:
                self.Delta_Lookup[Node] = i

        theta_edges = [edge.constraints[edge]["ctrl_high"]-edge.constraints[edge]["ctrl_low"]>0 for
             edge in self.Edges_list]

        self.n_nodes = len(self.Index_Lookup)
        self.n_theta = len(self.Theta_Lookup) + sum(theta_edges)
        self.n_delta = len(self.Delta_Lookup)

        return

    def Comp_edge_sets(self):
        """
        Compiles set of all possible edges, all existing edges and all edges possible to be added
        """
        All_edges = set()
        Existing_edges = set()
        Added_edges = set()

        for i in self.Index_Lookup.values():
            for j in range(i+1,max(self.Index_Lookup.values())+1):
                All_edges.add(frozenset([i,j]))
                Added_edges.add(frozenset([i,j]))


        for edge in self.Edges_list:
            conn = list(edge.connections)
```

```
                i = self.Index_Lookup[conn[0]]
                j = self.Index_Lookup[conn[1]]
                Existing_edges.add(frozenset([i,j]))
                Added_edges.remove(frozenset([i,j]))

            self.Conn_list = Existing_edges
            return All_edges,Existing_edges,Added_edges

        def Comp_Impedance_matrices(self, Slack_bus_connections = dict(), y_shunt = 0, M = 24):
            """
            Compile impedance matrices based on the grid and the slack bus admittance

            Slack_bus_connections: dictionary
                Slack bus connections to be used in the impedance matrix calculations with Node types as
                    key values and admittances as values
            M: int [Not used for final report]
                Maximum number of time instances to be used during OPF optimization
            """

            Y = np.zeros((1+self.n_nodes,1+self.n_nodes),dtype=complex)

            Y += np.eye(1+self.n_nodes)*y_shunt*1j


            Slack_node = 0
            for key in Slack_bus_connections.keys():
                i = Slack_node
                j = self.Index_Lookup[key]+1
                Y[i,j] = -Slack_bus_connections[key]
                Y[j,i] = -Slack_bus_connections[key]

                Y[i,i] = Slack_bus_connections[key]
                Y[j,j] = Slack_bus_connections[key]



            for Node in self.Nodes_list:
                for Edge in Node.connections:
                    for Destination in Edge.connections:
                        i = self.Index_Lookup[Node]+1
                        j = self.Index_Lookup[Destination]+1

                        if j!=i:
                            Y[i,j] = -Edge.admittance
                            Y[i,i] += Edge.admittance


            Y = Y[1:,1:]
            self.Y = Y
            try:
                Y_inv = np.linalg.inv(Y)
            except:
                print("Admittance matrix singular, SVD attempted")
                delt = 10**-8

                V,Lambda,V_T = np.linalg.svd(Y)
                for i in range(self.n_nodes):
                    Lambda[i] = 1/Lambda[i] if abs(Lambda[i]) >= delt else 0
                Y_inv = V@np.diag(Lambda)@V_T




            self.Z_p = Y_inv.real
            self.Z_q = Y_inv.imag


            n = self.n_nodes
            Z_P_Tilde = np.zeros((n*M,n*M))
            Z_Q_Tilde = np.zeros((n*M,n*M))



            for i in range(1,M+1):
                for j in range(i,i+1): #range(1:i+1) for lower triangular
                    Z_P_Tilde[(i-1)*n:i*n,(j-1)*n:j*n] = self.Z_p
                    Z_Q_Tilde[(i-1)*n:i*n,(j-1)*n:j*n] = self.Z_q

            self.Z_p_Tilde = Z_P_Tilde
            self.Z_q_Tilde = Z_Q_Tilde

            return self.Z_p, self.Z_q



        def Comp_Dist(self,Nodes):
            """
            Compile matrix of distances between pairs of nodes

            Nodes: Array
```

```python
            Array of node indices to be used for distance calculation
        """
        Max_Dist = ((max([nod.position[0] for nod in self.Nodes_list])-min([nod.position[0] for nod in self.Nodes_list]))**2 +\
                    (max([nod.position[1] for nod in self.Nodes_list])-min([nod.position[1] for nod in self.Nodes_list]))**2)**0.5
                    #For normalization

        distances = np.zeros((Nodes.shape[0]))

        for i in range(Nodes.shape[0]):
            distances[i] = ((self.Nodes_list[Nodes[i,0]].position[0]-self.Nodes_list[Nodes[i,1]].position[0])**2 + \
                            (self.Nodes_list[Nodes[i,0]].position[1]-self.Nodes_list[Nodes[i,1]].position[1])**2)**0.5

        return distances*self.Scale/Max_Dist

    def Comp_tree(self):
        """
        [Not used for final report]
        Calculate the number of connections for all nodes in the graph
        """
        Adjacency = np.array([len(node.connections) for node in self.Nodes_list])
        return Adjacency

    def Copy(self):
        """
        Copies self (deep copy)
        """
        copied = copy.deepcopy(self)
        return copied

    def Cluster(self, Samples = None, Cost_vector = None):
        """
        [Not used for final report]
        Cluster self, samples and cost vector by mergin leaf nodes onto their only neighbour.

        Samples: numpy array
            Samples to be clustered
        Cost_vector: numpy array
            OPF cost vector to be clustered
        """
        if self.n_nodes <= 2:
            raise MergeError("Minimum of 3 nodes required for merging")


        Clustered_Graph = self.Copy()

        Children_dict = dict()

        for i,node in enumerate(Clustered_Graph.Nodes_list):
            Children_dict[node] = self.Nodes_list[i]
            node.Clustered = False


        index1 = 0
        Merged_dict = dict()

        while index1<len(Clustered_Graph.Nodes_list):
            node1 = Clustered_Graph.Nodes_list[index1]

            Adjacency = Clustered_Graph.Comp_tree()
            Clustered_Graph.Comp_Lookups()
            Clustered_Graph.Comp_Impedance_matrices(Clustered_Graph.Slack_bus_connections)

            if Adjacency[index1] == 1 and node1 not in Clustered_Graph.Slack_bus_connections.keys():

                edge, = node1.connections
                for node2 in edge.connections:
                    if node1 != node2:
                        try:

                            admittance = Clustered_Graph.Y[Clustered_Graph.Index_Lookup[node2], Clustered_Graph.Index_Lookup[node2]]-\
                                         Clustered_Graph.Y[Clustered_Graph.Index_Lookup[node1], Clustered_Graph.Index_Lookup[node1]]


                            merged_node = node1.Merge(node2,admittance)
                            Clustered_Graph.Nodes_list += [merged_node]

                            index2 = Clustered_Graph.Index_Lookup[node2]

                            if not(Cost_vector is None):
                                Mult_high = node1.constraints[node1]["ctrl_high"].real/admittance.real + node2.constraints[node2]["ctrl_high"].real
                                cost_value_real = (Cost_vector[index1]*node1.constraints[node1]["ctrl_high"].real/admittance.real +\
                                                   Cost_vector[index2]*node2.constraints[node2]["
```

```
                                                       ctrl_high"].real)/Mult_high

                                try:
                                    Mult_high = node1.constraints[node1]["ctrl_high"].imag/admittance.imag + node2.constraints[node2]["ctrl_high"].imag
                                    cost_value_imag = (Cost_vector[index1+Clustered_Graph.n_nodes]*node1.constraints[node1]["ctrl_high"].imag/admittance.imag +\
                                                       Cost_vector[index2+Clustered_Graph.n_nodes]*node2.constraints[node2]["ctrl_high"].imag)/Mult_high
                                except:
                                    cost_value_imag = 0

                                Cost_vector = np.delete(Cost_vector,[index1,index2,index1+Clustered_Graph.n_nodes,index2+Clustered_Graph.n_nodes])
                                Cost_vector = np.concatenate((Cost_vector[:len(Cost_vector)//2],
                                                              np.array([cost_value_real]),
                                                              Cost_vector[len(Cost_vector)//2:],
                                                              np.array([cost_value_imag])))



                            Clustered_Graph.Nodes_list.pop(max(index1,index2))
                            Clustered_Graph.Nodes_list.pop(min(index1,index2))
                            index1 = 0

                            if node2.Clustered:
                                Merged_dict[merged_node] = {Children_dict[node1],*Merged_dict[node2]}
                                del Merged_dict[node2]
                            else:
                                Merged_dict[merged_node] = {Children_dict[node1],Children_dict[node2]}


                        except MergeError:
                            if node1.distribution is None:
                                pass
                            elif isinstance(node1.distribution,list):
                                for dist in node1.distribution:
                                    dist.multiplier *= admittance
                            else:
                                node1.distribution.multiplier *= admittance

                            index1 += 1
                            if node1.Clustered:
                                Merged_dict[node1] = {*Merged_dict[node1]}
                            else:
                                Merged_dict[node1] = {Children_dict[node1]}
            else:
                index1 += 1
                if node1.Clustered:
                    Merged_dict[node1] = {*Merged_dict[node1]}
                else:
                    Merged_dict[node1] = {Children_dict[node1]}


        Edges_set = set()
        for node in Clustered_Graph.Nodes_list:
            for conn in list(node.connections):
                Edges_set.add(conn)

        Clustered_Graph.Edges_list = list(Edges_set)

        Clustered_Graph.Comp_Lookups()
        Clustered_Graph.Comp_edge_sets()
        Clustered_Graph.Comp_Impedance_matrices(Clustered_Graph.Slack_bus_connections)

        if not(Samples is None):
            N,M = Samples.shape[0],Samples.shape[1]
            Clustered_Samples = np.zeros((N,M,Clustered_Graph.n_nodes),dtype=complex)

            for i,node in enumerate(Clustered_Graph.Nodes_list):
                index_list = [self.Index_Lookup[nod] for nod in list(Merged_dict[node])]
                Clustered_Samples[:,:,i] = np.sum(Samples[:,:,index_list],axis=2)
        else:
            Clustered_Samples = None


        return Clustered_Graph,Merged_dict,Clustered_Samples,Cost_vector


    def MultiSample(self,N,M = 1):
        """
        Multisample all random elements of nodes in the grid

        N: int
            Number of samples to be drawn
        M: int [Not used for final report]
```

```
            Number of time instances
        """
        Multi_S = np.zeros((N,M,self.n_nodes),dtype=complex)

        for Node in self.Delta_Lookup:
            Multi_S[:,:,self.Delta_Lookup[Node]] = Node.MultiSample(N,M)
        return Multi_S
```

CalcTools

```
import numpy as np
from scipy.special import comb as comb
from scipy.optimize import root_scalar


def Calc_eps(N,k,R,beta):
    """
    Calculate epsilon value with the number of samples, number of support constraints, number of discarded scenarios and confidence level

    N:     int
        Number of samples
    k:     int
        Number of support constraints
    R:     int
        Number of discarded scenarios
    beta: float
        beta value in (0,1) defining the confidence on the epsilon level
    """
    # Root-finder
    Comb1 = comb(N,k)
    Comb2 = comb(N+R,R)
    fun = lambda epsilon: (beta/(N+1))*sum([Comb1*(1-epsilon)**(m-k) for m in range(k,N) ]) - Comb2*Comb1*(1-epsilon)**(N-k)
    margin = 0.0000001
    bound_master = [0,1-margin] #Smaller search area to improve performance

    res = root_scalar(fun,x0=bound_master[0],x1=bound_master[1]/2,bracket = bound_master,xtol=margin)
    if res.converged:
        return res.root
    else:
        print(res)

def Calc_N(n_theta,epsilon,beta,method = "log"):
    """
    Calculate the prerequisite number of samples

    n_theta: int
        Number of control variable
    epsilon: float
        epsilon value in (0,1) defining scenario-based upper bound on violation probability
    beta:     float
        beta value in (0,1) defining the confidence on the epsilon level

    method:  string
        method of calculating the number of samples (default: log)
    """
    if method == "log":
        N = np.ceil(np.log(1/beta)*2/epsilon + 2*n_theta + np.log(2/epsilon)*2*n_theta/epsilon)
    elif method == "gen":
        # Not implemented
        raise NotImplementedError
    elif method == "lin":
        N = np.ceil(self.n_theta/(beta*epsilon)-1)

    return int(N)

def beta(x,alpha,beta):
    """
    Defines beta distribution

    x:      Array
        Array of values to calculate beta pdf for
    alpha: float
        alpha value for beta distribution
    beta:  float
        beta value for beta distribution
    """
    return (x**(alpha-1))*((1-x)**(beta-1))

def Eps_sieve_array_generator(N,beta,n_theta):
    """
    Calculate all a-posteriori epsilon values based on N, beta and n_theta

    N:     int
        Number of samples
```

```
    beta:     float
        beta value in (0,1) defining the confidence on the epsilon level
    n_theta: int
        Number of control variable
    """
    Eps_sieve_array = np.zeros((n_theta+1,n_theta*2))
    #assuming discarding 2*n_theta scenarios will have a-post epsilon >= a-priori epsilon based on samples
    for k in range(n_theta+1):
        for R in range(n_theta*2): #2*n_theta for guarantee that all R value have element in this array
            Eps_sieve_array[k,R] = Calc_eps(N,k,R,beta)

    return Eps_sieve_array


def Improvement_Array_generator(N,n_theta,k_orig = None,dpoints=5000):
    """
    Generates an array of the improvement probability of all combinations of support contraints k for 0-n_theta
        Where j (first axis) is the number of support constraints in the original optimization,
        and i (second axis) is the number of support constraints in the new optimization,
        with the elements being the improvement probability

    N:         int
        Number of samples
    n_theta: int
        Number of control variables

    k_orig:  int
        Number of original supporting constraints to limit number of calculations made, None for calculating entire array (default: None)
    dpoints: int
        Number of points to calculate beta pdf for (default: 5000)
    """

    Improvement_Array = np.zeros((n_theta+1,n_theta+1))

    x = np.linspace(0,1,dpoints)

    for i in range(0,n_theta+1):

        if k_orig == None:
            beta_i = beta(x,i+1,N-i+1)
            beta_i /= sum(beta_i)
            for j in range(i+1,n_theta+1):

                beta_j = beta(x,j+1,N-j+1)
                beta_j /= sum(beta_j)
                res = sum([beta_i[k]*(1-sum(beta_j[:k])) for k in range(dpoints-1)])
                Improvement_Array[j,i] = res

        else:
            if k_orig==i:
                beta_i = beta(x,i+1,N-i+1)
                beta_i /= sum(beta_i)
                for j in range(0,n_theta+1):
                    if j==i:
                        Improvement_Array[j,i] = 0.5
                        continue

                    beta_k = beta(x,j+1,N-j+1)
                    beta_k /= sum(beta_k)
                    res = sum([beta_k[k]*(1-sum(beta_i[:k])) for k in range(dpoints-1)])
                    Improvement_Array[i,j] = res

    if k_orig == None:
        Improvement_Array += np.triu(1-Improvement_Array.transpose(),k=1)
        Improvement_Array += np.eye(n_theta+1)/2

    return Improvement_Array



def Eps_Array_Generator(N,n_theta,beta):
    """
    Generates an array of the relative improvement in terms of a-posteriori epsilon values of all combinations of support contraints k for 0-n_theta
        Where j (first axis) is the number of support constraints in the original optimization,
        and i (second axis) is the number of support constraints in the new optimization,
        with the elements being the relative improvement

    N:         int
        Number of samples
    n_theta: int
        Number of control variables
    beta:  float
        beta value for beta distribution
    """

```

```python

    Eps_list = []
    for k in range(n_theta+1):
        Eps_list += [Calc_eps(N,k,0,beta)]

    Eps_array = np.zeros((n_theta+1,n_theta+1))
    for i in range(n_theta+1):
        for j in range(n_theta+1):
            Eps_array[i,j] = (Eps_list[i]-Eps_list[j])/Eps_list[j]


    return Eps_array/(Eps_array[n_theta,0])
```

GraphOPF

```python
import numpy as np
from GraphClass import *
from scipy import optimize as op

class OptimizationError(Exception):
    pass


def OPF_constraints(Graph,Samples, V_0 = 1, V_L = 1):
    """
    Build lower and upper bound based on samples and OPF function

    Graph: Graph
        Used to find voltage constraints
    Samples: Array
        Array of samples to compile bounds with

    V_0:     float
        Nominal voltage level (default: 1.0)
    V_L:     float
        Current voltage level (default: 1.0)
    """
    (N,M,n) = Samples.shape

    V_min = np.tile(np.array([Node.constraints[Node]["low"] for Node in Graph.Nodes_list]),M)
    V_max = np.tile(np.array([Node.constraints[Node]["high"] for Node in Graph.Nodes_list]),M)


    while M*n > Graph.Z_p_Tilde.shape[0]:
        print("Got here ",str(M))
        Graph.Comp_Impedance_matrices(Graph.Slack_bus_connections,M)



    Z_star = np.concatenate((Graph.Z_p_Tilde[:M*n,:M*n],Graph.Z_q_Tilde[:M*n,:M*n]),axis=1)

    lb = np.zeros((N,n*M))
    ub = np.zeros((N,n*M))

    for i in range(N):
        Sample = np.reshape(Samples[i,:,:],(n*M))
        Sample_star = np.concatenate((Sample.real,Sample.imag),axis=0)


        lb[i,:] = V_0*(V_min-V_L)-Z_star@Sample_star
        ub[i,:] = V_0*(V_max-V_L)-Z_star@Sample_star

    return lb,ub

def Optimize(Graph, lb, ub, M = 1, obj_func = lambda x: sum(x), Cost_vector = None, x_start = None,
 ftol = None):
    """
    Optimize grid performance according to lower and upper bound and objective function

    Graph: Graph
        Used to find voltage constraints
    lb:        Array
        Array of lower bounds on voltage levels over grid
    ub:        Array
        Array of upper bounds on voltage levels over grid

    M: int [Not used for final report]
        Number of time instances for which the OPF must be run in parallel
    obj_func: function
        function to be minimized over power input at controllable nodes (default: the sum of all
            delivered power)
    Cost_vector: numpy array
        Cost vector to minimized over power input at controllable nodes as c^T theta
    x_start: numpy array
        Starting position of optimization
    """
    # https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html
```

```
    # https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.LinearConstraint.html#
        scipy.optimize.LinearConstraint


    #OPF_constraints
    lb_limiting = np.max(lb,0)
    ub_limiting = np.min(ub,0)

    Z_star = np.concatenate((Graph.Z_p_Tilde[:Graph.n_nodes*M,:Graph.n_nodes*M],Graph.Z_q_Tilde[:
        Graph.n_nodes*M,:Graph.n_nodes*M]),axis=1)

    const_OPF = op.LinearConstraint(Z_star,lb_limiting,ub_limiting)

    #control_constraints
    lb_ctrl = np.zeros(Graph.n_nodes*2*M)
    ub_ctrl = np.zeros(Graph.n_nodes*2*M)

    for ctrl in Graph.Theta_Lookup:
        index = Graph.Theta_Lookup[ctrl]
        for i in range(M):
            lb_ctrl[index+i*Graph.n_nodes] = ctrl.constraints[ctrl]["ctrl_low"].real
            lb_ctrl[index+(i+M)*Graph.n_nodes] = ctrl.constraints[ctrl]["ctrl_low"].imag

            ub_ctrl[index+i*Graph.n_nodes] = ctrl.constraints[ctrl]["ctrl_high"].real
            ub_ctrl[index+(i+M)*Graph.n_nodes] = ctrl.constraints[ctrl]["ctrl_high"].imag


    const_ctrl = op.LinearConstraint(np.eye(Graph.n_nodes*2*M),lb_ctrl,ub_ctrl)

    #optimization
    if x_start is None:
        x0 = np.zeros(Graph.n_nodes*2*M)
    else:
        x0 = x_start

    if Cost_vector is None:
        res = op.minimize(obj_func,x0,constraints=(const_OPF,const_ctrl),tol = ftol)
    else:
        func = lambda x: obj_func(x,Cost_vector)
        res = op.minimize(func,x0,constraints=(const_OPF,const_ctrl),tol = ftol)

    res.fun/= M
    return res


def Limiting_constraints(Graph, lb, ub, V_L = 1 ):
    """Most limiting scenarios for each voltage level at each node

    Graph: Graph
        Graph used to provide constraints
    lb:  Array
        Array of lower bounds on voltage levels over grid
    ub:  Array
        Array of upper bounds on voltage levels over grid

    V_L: float
        Voltage level after optimization
    """
    lower_distance = V_L-lb
    upper_distance = ub-V_L

    delt = V_L*10**-6

    lower_limiting_index = np.argmin(lower_distance,axis=0)
    upper_limiting_index = np.argmin(upper_distance,axis=0)

    lower_limiting_index = lower_limiting_index[(lower_distance[lower_limiting_index]<=delt).any(
        axis=1)]
    upper_limiting_index = upper_limiting_index[(upper_distance[upper_limiting_index]<=delt).any(
        axis=1)]

    return np.unique(np.concatenate((lower_limiting_index,upper_limiting_index)))


def Support_constraints(Graph, lb, ub, M = 1, **kwargs_opt):
    """
    Check if sieving limiting constraints actually leads to "measurable" (delt) change in function
        value

    Graph: Graph
        Graph used to provide constraints
    lb: Array
        Array of lower bounds on voltage levels over grid
    ub: Array
        Array of upper bounds on voltage levels over grid

    M: int [Not used for final report]
        Number of time instances for which the OPF must be run in parallel

    **kwargs_opt:
```

```
        Arguments to be passed in optimization function
    """

    res = Optimize(Graph,lb,ub,M,**kwargs_opt)
    Z_star = np.concatenate((Graph.Z_p_Tilde[:Graph.n_nodes*M,:Graph.n_nodes*M],Graph.Z_q_Tilde[:Graph.n_nodes*M,:Graph.n_nodes*M]),axis=1)
    Limiting_scenarios = Limiting_constraints(Graph,lb,ub,Z_star@res.x)

    Supporting_scenarios = np.zeros((Limiting_scenarios.shape[0],2),dtype=float)
    Supporting_scenarios[:,0] = Limiting_scenarios

    for row,index in enumerate(Limiting_scenarios):
        #Exclude scenario that is limiting, report changed function value
        lb_temp = np.concatenate((lb[:index,:],lb[index+1:,:]))
        ub_temp = np.concatenate((ub[:index,:],ub[index+1:,:]))

        Supporting_scenarios[row,1] = Optimize(Graph,lb_temp,ub_temp,M,**kwargs_opt).fun - res.fun

    #Only pass (sorted) improving scenarios
    delt = res.fun*10**-8 #Minimal fractional improvement


    Supporting_scenarios = Supporting_scenarios[Supporting_scenarios[:,1]<-delt]
    Supporting_scenarios = Supporting_scenarios[np.argsort(Supporting_scenarios[:,1])]

    return Supporting_scenarios[:,0].astype(int)


def Sieve_constraints_optimize(Graph, Samples, epsilon, beta, Eps_sieve_array = None, **kwargs_opt):
    """
    Optimize performance upto specified level of epsilon by sieving constraints

    Graph: Graph
        Graph used to provide constraints
    Samples: Array
        Array of samples to compile bounds with
    epsilon: float
        epsilon value in (0,1) defining scenario-based upper bound on violation probability
    beta:    float
        beta value in (0,1) defining the confidence on the epsilon level

    Eps_sieve_array: numpy array
        Numpy array consisting of all a-posteriori epsilon for (k,R), values based on N, beta and n_theta

    **kwargs_opt:
        Arguments to be passed in optimization function
    """

    M = Samples.shape[1]

    N = CalcTools.Calc_N(Graph.n_theta*M,epsilon,beta)
    if Samples.shape[0] < N:
        raise ValueError("Not enough samples for specified levels of epsilon and beta; \nSamples: "+str(Samples.shape[0])+", Required: "+str(N))
    Sieved = set()

    if Eps_sieve_array is None:
        Eps_sieve_array = CalcTools.Eps_sieve_array_generator(N,beta,Graph.n_theta*M)


    lb,ub = OPF_constraints(Graph,Samples)

    res = Optimize(Graph,lb,ub,M,**kwargs_opt)

    lb_sieved,ub_sieved = OPF_constraints(Graph,Samples)
    if res.success:
        Sieved_more = 1
        while Sieved_more>0:

            Supporting_scenarios = Support_constraints(Graph,lb_sieved,ub_sieved,M,**kwargs_opt)
            k = min(Supporting_scenarios.shape[0],Graph.n_theta)

            R = len(Sieved)

            max_R = len(Eps_sieve_array[k,Eps_sieve_array[k,:] <= epsilon])-1

            if max_R == R:
                break

            Sieved_more = min(max_R-R,k)
            sieved_scen = Supporting_scenarios[:Sieved_more]

            for scen in sieved_scen:
                lb_sieved = np.concatenate((lb_sieved[:scen,:],lb_sieved[scen+1:,:]))
                ub_sieved = np.concatenate((ub_sieved[:scen,:],ub_sieved[scen+1:,:]))


            for siev in sorted(Sieved):
                sieved_scen += siev <= sieved_scen
```

```
                for scen in sieved_scen:
                    Sieved.add(scen)

            res = Optimize(Graph,lb_sieved,ub_sieved,M,**kwargs_opt)
            eps = CalcTools.Calc_eps(N,k,len(Sieved),beta)


            return res,Sieved,eps
        raise OptimizationError ("infeasible realization; "+str(res))
        return 0


def Monte_Carlo_optimize(Graph,Samples,**kwargs_opt):
    """
    Optimize performance using the Monte-Carlo approach

    Graph: Graph
        Graph used to provide constraints
    Samples: Array
        Array of samples to compile bounds with

    **kwargs_opt:
        Arguments to be passed in optimization function
    """
    lb,ub = OPF_constraints(Graph,Samples)

    N = Samples.shape[0]
    M = Samples.shape[1]

    res_list = np.empty((N,2))
    x_list = np.empty((N,Graph.n_nodes*2*M))


    x_start = None
    update_freq = 10

    for i in range(0,N):
        if i%update_freq == 2:
            x_start = np.average(x_list[:i,:],axis=0)
        res = Optimize(Graph,np.expand_dims(lb[i,:],axis=0), np.expand_dims(ub[i,:],axis=0), M,
             x_start = x_start, **kwargs_opt)
        x_list[i,:] = res.x
        res_list[i,0] = res.fun
        res_list[i,1] = res.success


    return x_list,res_list

def Single_step(Graph, Sample, x, V_0 = 1, V_L = 1):
    """
    Compute voltage balance levels for a combination of a sample and an input vector

    Graph: Graph
        Graph used to provide power dynamics
    Samples: Array
        Array of sampled loads
    x: array
        Array of controlled loads

    V_0:     float
        Nominal voltage level (default: 1.0)
    V_L:     float
        Current voltage level (default: 1.0)
    """
    (M,n) = Sample.shape
    Sample = np.reshape(Sample,(n*M))
    Sample_star = np.concatenate((Sample.real,Sample.imag),axis=0)
    Z_star = np.concatenate((Graph.Z_p_Tilde[:M*n,:M*n],Graph.Z_q_Tilde[:M*n,:M*n]),axis=1)

    return (V_L + Z_star@(Sample_star+x)/V_0).reshape(M,Graph.n_nodes)
```

GraphOptimizationLoop

```
import numpy as np
import cv2 as cv
import datetime

from GraphClass import *
import CalcTools
import GraphOPF
import GraphPlot



def Find_all_edge_mod(Graph, Samples, epsilon, beta, Eps_sieve_array, ViolProb_array, *args_edge,
     Graph_basecase = None, Weights = np.array([100,100,10,5]),  kwargs_OPF={}, kwargs_opt={},
     Early_prune=False, Cutoff = 0, Budget = 10, Selected_edges = None, method = "Monte Carlo"):
```

```
    """
    Function that finds all updates to edges on the graph, including both upgrading existing edges and adding new ones

    Graph:          Graph
        Graph from Graph function file consisting of nodes and edges
    Samples:        array
        Samples used in optimization
    epsilon:        float
        epsilon value in (0,1) defining scenario-based upper bound on violation probability
    beta:           float
        beta value in (0,1) defining the confidence on the epsilon level
    Eps_sieve_array:numpy array
        Numpy array consisting of all a-posteriori epsilon for (k,R), values based on N, beta and n_theta
    ViolProb_array: array
        Array of the improvement probability of all combinations of support contraints k for 0-n_theta

    *args_edge:
        Arguments to be passed into new edge, first one to be the admittance value

    Graph_basecase: Graph
        Graph used as a basecase to compare against, used for larger horizon depth
    Weights:        array
        Weights to be used in optimization in the order:
            1) Probability of improvement of the violation probability
            2) Function value improvement
            3) Distance covered by new edge
            4) Distance covered by existing edge

    kwargs_OPF:
        Arguments to be passed into optimal power flow model resulting in lower bounds and upper bounds on control inputs
    kwargs_opt:
        Arguments to be passed in optimization function

    Early_prune:    bool [Not used for final report]
        Discard optional changes if new addition is no longer able to surpass current best option
    Cutoff:         float
        Score value where optimization is terminated
    Budget:         float
        Stopping condition, if the total cost of all implemented modifications is above this value, the loop is terminated
    Selected_edges: set [Not used for final report]
        Set of all edges considered promising (mainly used in clustering)
    method:         string
        Method used for grid expansion optimization. Either 'Scenario' or 'Monte Carlo' (default)
    """
    M = Samples.shape[1]

    _,Upgraded_edges,Added_edges = Graph.Comp_edge_sets()
    Added_edges_list = np.array([[min(el1,el2),max(el1,el2)] for el1,el2 in list(Added_edges)])
    Upgraded_edges_list = np.array([[min(el1,el2),max(el1,el2)] for el1,el2 in list(Upgraded_edges)])

    # 2 edge cases: upgraded edges empty & added edges empty (smaller grids more likely)

    Scoring_array = np.zeros((1,3))
    if not(Added_edges_list.shape == (0,)):
        Scoring_array = np.concatenate((Scoring_array,\
                                        np.concatenate((np.ones((Added_edges_list.shape[0],1)), Added_edges_list),axis=1),\
                                       ),axis=0)

    if not(Added_edges_list.shape == (0,)):
        Scoring_array = np.concatenate((Scoring_array,\
                                        np.concatenate((np.zeros((Upgraded_edges_list.shape[0],1)), Upgraded_edges_list),axis=1),\
                                       ),axis=0)


    Scoring_array = np.concatenate((np.zeros((Scoring_array.shape[0],1)),
                                    Scoring_array,np.zeros((Scoring_array.shape[0],12-Scoring_array.shape[1])))),axis=1)

    # Scoring array:
    # #modifications + 1 x 12
    # 0: Index
    # 1: Addition (1) or modification (0)
    # 2: Node 1 associated with edge
    # 3: Node 2 associated with edge
    # 4: Distance between nodes
    # 5: Robustness metric
    # 6: Operational performance
    # 7: Score associated with cost of modification
    # 8: Score associated with robustness improvement
    # 9: Score associated with operational performance
    # 10: Total score (direct)
    # 11: Fully explored (1) Not fully explored (0)
```

```
    #
    # First row --> basecase
    # All other rows --> modifications


    # Basecase
    if Graph_basecase is None:
        Graph_basecase = Graph.Copy()

    if method == "Scenario":
        lb,ub = GraphOPF.OPF_constraints(Graph_basecase, Samples,**kwargs_OPF)
        supp_basic = GraphOPF.Support_constraints(Graph_basecase,lb,ub,M,**kwargs_opt)
        res,_,_ = GraphOPF.Sieve_constraints_optimize(Graph_basecase, Samples, epsilon, beta, Eps_sieve_array, **kwargs_opt)

        Scoring_array[0,5] = len(supp_basic)
        Scoring_array[0,6] = res.fun

    elif method == "Monte Carlo":
        _,res_list = GraphOPF.Monte_Carlo_optimize(Graph_basecase,Samples,**kwargs_opt)
        Scoring_array[0,5] = 1-np.average(res_list[:,1])
        Scoring_array[0,6] = np.average(res_list[res_list[:,1]==1,0])
    else:
        raise KeyError ("Incorrect method given")

    Scoring_array[0,10] = Cutoff
    Scoring_array[0,11] = 1

    Scoring_array[1:,4] = Graph.Comp_Dist(Scoring_array[1:,[2,3]].astype(int))
    Scoring_array[1:,7] += (Scoring_array[1:,1]*Weights[2] + (1-Scoring_array[1:,1])*Weights[3])*Scoring_array[1:,4]

    for i in range(1,Scoring_array.shape[0]):
        Scoring_array[i,0] = i



        if Scoring_array[i,1]:

            # Check if current addition is between two nodes of interest. Check not done for upgrades
            if Selected_edges != None:
                begin,end = Scoring_array[i,[2,3]].astype(int)
                if Graph.Nodes_list[begin] not in Selected_edges or Graph.Nodes_list[end] not in Selected_edges:
                    continue


            #Added
            begin,end = Scoring_array[i,[2,3]].astype(int)
            edge = Graph.Add_edge(Graph.Nodes_list[begin],Graph.Nodes_list[end],*args_edge)

        else:
            #Updated
            for edge in Graph.Edges_list:
                begin,end = edge.connections
                if set([Graph.Index_Lookup[begin],Graph.Index_Lookup[end]]) == set(Scoring_array[i,[2,3]]):
                    break
            admittance = args_edge[0]
            admittance_update = edge.admittance + admittance
            Graph.Update_edge(edge, new_admittance = admittance_update)



        if method == "Scenario":
            if not(Scoring_array[i,7] > Budget and Early_prune):
                Graph.Comp_Impedance_matrices(Graph.Slack_bus_connections)
                try:
                    if not(Scoring_array[i,7] - sum(Weights[[0,1]]) > Cutoff and Early_prune):
                        lb,ub = GraphOPF.OPF_constraints(Graph,Samples,**kwargs_OPF)
                        supp = GraphOPF.Support_constraints(Graph,lb,ub,M,**kwargs_opt)
                        Scoring_array[i,5] = len(supp)

                        # Violprob array based on a-posteriori epsilon
                        # Scoring_array[i,8] += -ViolProb_array[int(Scoring_array[0,5]),min(int(Scoring_array[i,5]),Graph.n_theta)]*Weights[0]

                        # Violprob array based on probability density functions
                        Scoring_array[i,8] += (0.5-ViolProb_array[int(Scoring_array[0,5]),min(int(Scoring_array[i,5]),Graph.n_theta)])*2*Weights[0]

                        if not(Scoring_array[i,7] + Scoring_array[i,8] - sum(Weights[[1]]) > Cutoff and Early_prune): #assumes that the performance is >= 0 always
                            res,_,_ = GraphOPF.Sieve_constraints_optimize(Graph, Samples, epsilon, beta, Eps_sieve_array, **kwargs_opt)
                            Scoring_array[i,6] = res.fun
                            if Scoring_array[0,6] == 0.0:
                                Scoring_array[i,9] += (Scoring_array[i,6]-Scoring_array[0,6])*Weights[[1]]
```

```python
                            else:
                                Scoring_array[i,9] += ((Scoring_array[i,6]-Scoring_array[0,6])/Scoring_array[0,6])*Weights[[1]]

                            Scoring_array[i,10] = np.sum(Scoring_array[i,[7,8,9]],axis=0)
                            Scoring_array[i,11] = 1
                except GraphOPF.OptimizationError:
                    pass


        elif method == "Monte Carlo":
            if not(Scoring_array[i,7] - sum(Weights[[0,1]]*np.array([Scoring_array[0,5]!=0.,1]))) > Cutoff and Early_prune):
                Graph.Comp_Impedance_matrices(Graph.Slack_bus_connections)
                _,res_list = GraphOPF.Monte_Carlo_optimize(Graph,Samples,**kwargs_opt)

                Scoring_array[i,5] = 1-np.average(res_list[:,1])
                Scoring_array[i,6] = np.average(res_list[res_list[:,1]==1,0])

                # [] Correct scaling for relative improvement when original value was 0 --> currently no scaling but should be >> 1
                if Scoring_array[0,5] == 0.0:
                    Scoring_array[i,8] += (Scoring_array[i,5]-Scoring_array[0,5])*Weights[0]
                else:
                    Scoring_array[i,8] += ((Scoring_array[i,5]-Scoring_array[0,5])/Scoring_array[0,5])*Weights[[0]]

                if Scoring_array[0,6] == 0.0:
                    Scoring_array[i,9] += (Scoring_array[i,6]-Scoring_array[0,6])*Weights[[1]]

                else:
                    Scoring_array[i,9] += ((Scoring_array[i,6]-Scoring_array[0,6])/Scoring_array[0,6])*Weights[[1]]

                Scoring_array[i,10] = np.sum(Scoring_array[i,[7,8,9]],axis=0)
                Scoring_array[i,11] = 1
        else:
            raise KeyError ("Incorrect method given")

        Cutoff = np.min(Scoring_array[:,10])

        if Scoring_array[i,1]:
            Graph.Remove_edge(edge)
        else:
            Graph.Update_edge(edge, new_admittance = admittance_update-admittance)


    Scoring_array = Scoring_array[Scoring_array[:,11]==1]



    return Scoring_array[1:,:11]




def Depth_levels_recursive(Graph, *args_edge, depth = 0):
    """
    Build a tree of all possible implementations of combinations of modifications

    Graph: Graph
        Graph were all implementations should be passed onto
    *args_edge:
        Arguments to be passed into new edge, first one to be the admittance value
    depth: int
        Size of the combinations of modifications
    """
    if depth == 0:
        return [Graph]
    else:
        Copied_graphs_list = []
        _,Upgraded_edges,Added_edges = Graph.Comp_edge_sets()

        for u in list(Upgraded_edges):
            begin,end = u
            Copied_g = Graph.Copy()

            for edge in Copied_g.Edges_list:
                begin_node,end_node = edge.connections
                if set([Copied_g.Index_Lookup[begin_node],Copied_g.Index_Lookup[end_node]]) == set([begin,end]):
                    admittance = args_edge[0]
                    admittance_update = edge.admittance + admittance
                    Copied_g.Update_edge(edge, new_admittance = admittance_update)
                    if depth == 1:
                        Copied_graphs_list += [(Copied_g,edge)]
                    else:
                        Copied_graphs_list += [(Depth_levels_recursive(Copied_g,*args_edge,depth = depth-1),Copied_g,edge)]
                    break
```

```python

        for a in list(Added_edges):
            begin,end = a
            Copied_g = Graph.Copy()

            edge = Copied_g.Add_edge(Copied_g.Nodes_list[begin],Copied_g.Nodes_list[end],*args_edge)
            if depth == 1:
                Copied_graphs_list += [(Copied_g,edge)]
            else:
                Copied_graphs_list += [(Depth_levels_recursive(Copied_g,*args_edge,depth = depth-1),Copied_g,edge)]

        return Copied_graphs_list




def Find_all_edge_mod_recursive(Graph, Samples, epsilon, beta, Eps_sieve_array, ViolProb_array, *args_edge, Graph_basecase = None, Weights = np.array([100,100,10,5]), Branch_depth = 1, Branch_breadth = 4, kwargs_OPF={}, kwargs_opt={}, Early_prune=False, Cutoff = 0, Budget = 10, Selected_edges = None, method = "Monte Carlo"):
    """
    Function that finds combinations of modifications, including both upgrading existing edges and adding new ones.
    For each step further into the future, only the top 'Branch_breadth' modifications on the previous branch are candidates to be inspected further, until depth 'Branch_depth' is reached or other stopping condition is met.
    Ultimately, the modification with the best score on the development horizon is picked to be installed

    Graph:          Graph
        Graph from Graph function file consisting of nodes and edges
    Samples:        array
        Samples used in optimization
    epsilon:        float
        epsilon value in (0,1) defining scenario-based upper bound on violation probability
    beta:           float
        beta value in (0,1) defining the confidence on the epsilon level
    Eps_sieve_array:numpy array
        Numpy array consisting of all a-posteriori epsilon for (k,R), values based on N, beta and n_theta
    ViolProb_array: array
        Array of the improvement probability of all combinations of support contraints k for 0-n_theta

    *args_edge:
        Arguments to be passed into new edge, first one to be the admittance value

    Graph_basecase: Graph
        Graph used as a basecase to compare against, used for larger horizon depth
    Weights:        array
        Weights to be used in optimization in the order:
            1) Probability of improvement of the violation probability
            2) Function value improvement
            3) Distance covered by new edge
            4) Distance covered by existing edge
    Branch_depth:   int
        Depth of planning horizon; The number of modifications to be inspected at once
    Branch_breadth: int
        Breadth of planning horizon; The number of modifications for each branch that warrant further study

    kwargs_OPF:
        Arguments to be passed into optimal power flow model resulting in lower bounds and upper bounds on control inputs
    kwargs_opt:
        Arguments to be passed in optimization function

    Early_prune:    bool [Not used for final report]
        Discard optional changes if new addition is no longer able to surpass current best option
    Cutoff:         float
        Score value where optimization is terminated
    Budget:         float
        Stopping condition, if the total cost of all implemented modifications is above this value, the loop is terminated
    Selected_edges: set [Not used for final report]
        Set of all edges considered promising (mainly used in clustering)
    method:         string
        Method used for grid expansion optimization. Either 'Scenario' or 'Monte Carlo' (default)
    """

    Scoring_array = Find_all_edge_mod(Graph,
                                      Samples,
                                      epsilon,
                                      beta,
                                      Eps_sieve_array,
                                      ViolProb_array,
                                      *args_edge,
                                      Graph_basecase = Graph_basecase,
                                      Weights = Weights,
```

```
                                    kwargs_OPF = kwargs_OPF,
                                    kwargs_opt = kwargs_opt,
                                    Early_prune = False,
                                    Cutoff = Cutoff,
                                    Budget = Budget,
                                    method = method)

    Scoring_array = Scoring_array[np.argsort(Scoring_array[:,10])]

    Scoring_array = Scoring_array[Scoring_array[:,7] <= Budget]


    if Branch_depth <= 1:
        return Scoring_array
    else:
        Graph_depth = Depth_levels_recursive(Graph,*args_edge,depth=1)
        for row in Scoring_array[:Branch_breadth if Branch_breadth >= 1 else Scoring_array.shape[0],:]:
            for step in Graph_depth:
                Graph_copied, edge = step
                begin,end = edge.connections
                if set(row[[2,3]]) == set([Graph_copied.Index_Lookup[begin],Graph_copied.Index_Lookup[end]]):
                    break


            Scoring_array_next_step = Find_all_edge_mod_recursive(Graph_copied,
                                                                  Samples,
                                                                  epsilon,
                                                                  beta,
                                                                  Eps_sieve_array,
                                                                  ViolProb_array,
                                                                  *args_edge,
                                                                  Graph_basecase = Graph_basecase,
                                                                  Weights = Weights,
                                                                  Branch_depth = Branch_depth - 1,
                                                                  Branch_breadth = Branch_breadth,
                                                                  kwargs_OPF = kwargs_OPF,
                                                                  kwargs_opt = kwargs_opt,
                                                                  Early_prune = Early_prune,
                                                                  Cutoff = Cutoff,
                                                                  Budget = Budget - row[7],
                                                                  Selected_edges = Selected_edges,
                                                                  method = method)


            Scoring_array_next_step = Scoring_array_next_step[np.argsort(Scoring_array_next_step[:,10])]

            if Scoring_array_next_step.shape[0] == 0:
                row[10] = np.sum(row[[7,8,9]])
            else:
                row[10] = np.min(Scoring_array_next_step[:,10]) + row[7]

        return Scoring_array

def Optimization_loop(Graph_master, epsilon, beta, M, *args_edge, Samples = None, Weights=np.array([20,10,10,5]), ViolProb_array = None, Cutoff = 0, Budget = 10, node_max = -1, Leniancy = 0.05, Branch_depth = 1, Branch_breadth = 4, kwargs_OPF={}, kwargs_opt={}, method = "Scenario", Verbose=False, kwargs_plot={}, FileDir=None):
    """
    Graph_master:           Graph
        Graph from Graph function file consisting of nodes and edges to be optimized
    epsilon:        float
        epsilon value in (0,1) defining scenario-based upper bound on violation probability
    beta:           float
        beta value in (0,1) defining the confidence on the epsilon level
    M: int [Not used for final report]
            Number of time instances

    *args_edge:
        Arguments to be passed into new edge, first one to be the admittance value

    Samples:        array
        Samples used in optimization
    Weights:        array
        Weights to be used in optimization in the order:
            1) Probability of improvement of the violation probability
            2) Function value improvement
            3) Distance covered by new edge
            4) Distance covered by existing edge
    ViolProb_array: array
        Array of the improvement probability of all combinations of support contraints k for 0-n_theta
    Cutoff:         float
        Score value where optimization is terminated
    Budget:         float
        Stopping condition, if the total cost of all implemented modifications is above this value, the loop is terminated
    node_max:       int [Not used for final report]
```

```
        Target value for the number of nodes to cluster to. Disable clustering: -1 (default)
    leniancy:       float [Not used for final report]
        Growth rate (0,1] of stopping conditions 'Cutoff' and 'Budget' for each level of clustering. Lower value corresponds to a more leniant approach
    Branch_depth:   int
        Depth of planning horizon; The number of modifications to be inspected at once
    Branch_breadth: int
        Breadth of planning horizon; The number of modifications for each branch that warrant further study
    kwargs_OPF:
        Arguments to be passed into optimal power flow model resulting in lower bounds and upper bounds on control inputs
    kwargs_opt:
        Arguments to be passed in optimization function
    method:         string
        Method used for grid expansion optimization. Either 'Scenario' or 'Monte Carlo' (default)
    Verbose:        bool
        Defines if intermediate results will be printed
    kwargs_plot:
        Arguments to be passed into grid plots
    FileDir:        str
        Directory in which results will be saved, None for no information saved (default: None)
    """



    N = CalcTools.Calc_N(Graph_master.n_theta*M,epsilon,beta)
    if method == "Scenario":

        # Violprob array based on epsilon
        # ViolProb_array = CalcTools.Eps_Array_Generator(N,Graph_master.n_theta,beta)

        # Violprob array based on probability density functions
        if ViolProb_array is None:
            ViolProb_array = CalcTools.Improvement_Array_generator(N,Graph_master.n_theta*M)
        Eps_sieve_array = CalcTools.Eps_sieve_array_generator(N,beta,Graph_master.n_theta*M)
    else:
        ViolProb_array = None
        Eps_sieve_array = None

    if Samples is None:
        Samples = Graph_master.MultiSample(N,M)

    while True:
        try:
            GraphOPF.Sieve_constraints_optimize(Graph_master, Samples, epsilon, beta, Eps_sieve_array, **kwargs_opt)
            break
        except GraphOPF.OptimizationError:
            print("Infeasible sample, resampling")
            Samples = Graph_master.MultiSample(N,M)
            pass

    Upgrades_list = []

    Frame = GraphPlot.Draw_Graph(Graph_master,**kwargs_plot)
    if Verbose:
        print("\n ____________________ \n")
        print(datetime.datetime.now())
        print("Grid optimization: \n")
        cv.imshow("Grid",Frame)
        cv.waitKey(1)
    if FileDir != None:
        imgpath = FileDir+"/Grid__start.png"
        cv.imwrite(imgpath, Frame*255)
        # Save Grid parameters
        setpath = FileDir+"/Grid_parameters.txt"
        setfile = open(setpath, "w")
        setfile.write(Graph_master.Summarize())
        setfile.close()

        # Save Optimization settings
        Settings = {**kwargs_OPF,**kwargs_opt}
        Settings["method"]=method
        Settings["Cutoff"]=Cutoff
        Settings["Weights"]=Weights
        Settings["epsilon"]=epsilon
        Settings["beta"]=beta
        Settings["M"]=M
        Settings["node_max"]=node_max #[Not used for final report]
        Settings["Leniancy"]=Leniancy #[Not used for final report]
        Settings["Branch_depth"]=Branch_depth
        Settings["Branch_breadth"]=Branch_breadth

        setpath = FileDir+"/Optimization_settings.txt"
        setfile = open(setpath, "w")
        setfile.write(str(Settings)[1:-1])
        setfile.close()

        # Save Samples
```

```
        sampath = FileDir+"/Samples.txt"
        np.savetxt(sampath,np.reshape(Samples,(N,Graph_master.n_nodes*M)))

    start_time = datetime.datetime.now()


    # Used for clustering approach [Not used for final report]
    Graph_clustered_list = [Graph_master]
    parent_dict_list = [None]
    Samples_list = [Samples]

    if "Cost_vector" in kwargs_opt.keys():
        Cost_vector_list = [kwargs_opt["Cost_vector"]]
    else:
        Cost_vector_list = [None]


    while True:

        if Verbose:
            Frame = GraphPlot.Draw_Graph(Graph_master,**kwargs_plot)
            cv.imshow("Grid",Frame)
            cv.waitKey(1)

        # Loop used for clustering approach [Not used for final report]
        if node_max >= 0:
            if Verbose:
                print("Clustering graph, node goal: "+str(node_max))
            while True:

                if Verbose:
                    print("Current node count: "+str(Graph_clustered_list[-1].n_nodes))

                if Graph_clustered_list[-1].n_nodes <= node_max:
                    if Verbose:
                        print("Node goal attained, clustering complete \n")
                    break

                prev = Graph_clustered_list[-1].n_nodes
                try:
                    Graph, Parent_dict, Clust_samp, Cost_vector = Graph_clustered_list[-1].Cluster(Samples = Samples_list[-1])

                except MergeError:

                    Graph_clustered_list = Graph_clustered_list[:-1]
                    parent_dict_list = parent_dict_list[:-1]
                    Samples_list = Samples_list[:-1]
                    Cost_vector_list = Cost_vector_list[:-1]


                    if Verbose:
                        print("No improvement found, clustering abandonded \n")
                    break

                Graph_clustered_list += [Graph]
                parent_dict_list += [Parent_dict]
                Samples_list += [Clust_samp]
                Cost_vector_list += [Cost_vector]

                if Graph_clustered_list[-1].n_nodes == prev:

                    Graph_clustered_list = Graph_clustered_list[:-1]
                    parent_dict_list = parent_dict_list[:-1]
                    Samples_list = Samples_list[:-1]
                    Cost_vector_list = Cost_vector_list[:-1]

                    if Verbose:
                        print("No improvement found, clustering abandonded \n")
                    break


        # For lowest cluster layer n: Calculate all additions
        # Store all nodes associated upgrades with a lower score than Cutoff * u^n
        # move up cluster layer, calculate all additions only using the prev nodes
        # Calculate updates over all edges in master graph



        Selected_edges = None
        # Loop used for clustering approach, for report, loop is only run once for the original graph
        for i in range(1,len(Graph_clustered_list)+1):

            Graph = Graph_clustered_list[-i]
            Samples = Samples_list[-i]
            kwargs_opt_temp = {key:kwargs_opt[key] if key != "Cost_vector" else Cost_vector_list[-i] for key in kwargs_opt.keys()}


```

```
            if Verbose:
                Frame = GraphPlot.Draw_Graph(Graph,**kwargs_plot)
                cv.imshow("Grid",Frame)
                cv.waitKey(1)


            # Remove upgrades from solution space for clustered graphs [Not used for final report]
            w_temp = Weights if i == len(Graph_clustered_list) else Weights*np.array([1,1,1,10**16])

            # Change stopping criteria for clustered graph [Not used for final report]
            Cutoff_temp = Cutoff*Leniancy**(-i+1)
            Budget_temp = Budget*Leniancy**(-i+1)



            Scoring_array = Find_all_edge_mod_recursive(Graph,
                                                        Samples,
                                                        epsilon,
                                                        beta,
                                                        Eps_sieve_array,
                                                        ViolProb_array,
                                                        *args_edge,
                                                        Graph_basecase = Graph.Copy(),
                                                        Weights = Weights,
                                                        Branch_depth = Branch_depth,
                                                        Branch_breadth = Branch_breadth,
                                                        kwargs_OPF = kwargs_OPF,
                                                        kwargs_opt = kwargs_opt_temp,
                                                        Early_prune = False,
                                                        Cutoff = Cutoff_temp,
                                                        Budget = Budget_temp,
                                                        Selected_edges = Selected_edges,
                                                        method = method)


            Scoring_array = np.concatenate((Scoring_array,np.expand_dims(np.sum(Scoring_array
                 [:,[7,8,9]],axis=1),axis=1)),axis=1)

            Scoring_array = Scoring_array[np.argsort(Scoring_array[:,11])]
            Scoring_array = Scoring_array[np.argsort(Scoring_array[:,10])]

            Selected_Mod = Scoring_array[Scoring_array[:,10]<=Cutoff_temp]


            if i == len(Graph_clustered_list):
                break


            # Prepare promising edges for cluster level higher [Not used for final report]
            Selected_edges = set()
            for j in range(Selected_Mod.shape[0]):
                Selected_edges.update({*parent_dict_list[-i][Graph.Nodes_list[int(Selected_Mod[j,2])
                     ]],\
                                       *parent_dict_list[-i][Graph.Nodes_list[int(Selected_Mod[j,3])
                                            ]]})

            if Verbose:
                print(f"Cluster level {(len(Graph_clustered_list)-i):2.0f}")
                print(f"{len(Scoring_array):4.0f} Edge modifications inspected \n")
                print(f"{len(Selected_edges):4.0f} nodes of interest:")
                for nod in list(Selected_edges)[:3]:
                    print(nod)
                if len(list(Selected_edges))>3:
                    print("   ...  ({} more rows) ...".format(len(list(Selected_edges))-3))
                print()



        # Remove all modifications that exceed remaining budget
        while True:
            if Scoring_array.shape[0] >= 1 and Scoring_array[0,7]>Budget:
                Scoring_array = Scoring_array[1:,:]
            else:
                break


        if Scoring_array.shape[0] < 1:
            if Verbose:
                print("No (further) improvement found.\n")
            break

        if Verbose:
            print(datetime.datetime.now())
            print()
            print("  Add/Up  Node 1  Node 2  Score (horizon) Score (direct)")
            if Scoring_array.shape[0] <= 7:
                print(np.round(Scoring_array[:,[1,2,3,-2,-1]],3))
            else:
                print(np.round(Scoring_array[:5,[1,2,3,-2,-1]],3))
                print("   ...  ({} more rows) ...".format(Scoring_array.shape[0]-7))
```

```
                print(np.round(Scoring_array[-2:,[1,2,3,-2,-1]],3))
            print()

        if Scoring_array[0,11] <= Cutoff:

            if Scoring_array[0,1]:
                #Addition
                node_begin,node_end = Scoring_array[0,[2,3]].astype(int)
                edge = Graph_master.Add_edge(Graph_master.Nodes_list[node_begin],Graph_master.Nodes_list[node_end],*args_edge)

                cost = Scoring_array[0,4]*Weights[2]

                if Verbose:
                    print("Added an edge:")

            else:
                #Upgrade
                for edge in Graph_master.Edges_list:
                    begin,end = edge.connections
                    if set([Graph_master.Index_Lookup[begin],Graph_master.Index_Lookup[end]]) == set(Scoring_array[0,[2,3]].astype(int)):
                        break
                admittance = edge.admittance + args_edge[0]
                Graph_master.Update_edge(edge, new_admittance = admittance)

                cost = Scoring_array[0,4]*Weights[3]

                if Verbose:
                    print("Upgraded an edge:")

            begin,end = edge.connections

            time = (datetime.datetime.now()-start_time).total_seconds()

            Upgrades_list += [[Graph_master.Index_Lookup[begin],Graph_master.Index_Lookup[end],Scoring_array[0,1],edge.admittance,cost,time]]
            Graph_master.Comp_Impedance_matrices(Graph_master.Slack_bus_connections)

            Edge_col = edge.color
            edge.color = np.array([140,240,140],dtype=float)
            Frame = GraphPlot.Draw_Graph(Graph_master,**kwargs_plot)

            Budget -= cost

            if Verbose:
                print(edge)
                print("\nBudget remaining: {}".format(Budget))
                print("\n _______________________ \n")

                cv.imshow("Grid",Frame)
                cv.waitKey(2000)

            if FileDir != None:
                imgpath = FileDir+"/Grid_iteration_"+str(len(Upgrades_list))+".png"
                cv.imwrite(imgpath, Frame*255)

            edge.color = Edge_col

        else:
            if Verbose:
                print("No (further) improvement found.\n")

            break
        if Scoring_array.shape[0] <= 1:
            if Verbose:
                print("All improvements implemented.\n")
            break

    Upgrades_array = np.array(Upgrades_list,dtype=float)
    cv.destroyWindow("Grid")

    # Save results if required
    if FileDir != None:
        respath = FileDir+"/Edges_added.txt"
        np.savetxt(respath,Upgrades_array)
        if Verbose:
            print("Results and settings successfully saved to directory:\n"+FileDir)

    return Upgrades_array,Graph_master.Copy()
```

GraphValidation

```
import numpy as np
import matplotlib.pyplot as plt

from GraphClass import *
```

```
import GraphOPF
import CalcTools


def Single_run_scenario(Graph, epsilon = 0, beta = 0, sieving = True, Samples = None, kwargs_opt={}):
    """
    Run a single OPF optimization using the scenario approach

    Graph: Graph
        Graph used to provide constraints

    epsilon: float
        epsilon value in (0,1) defining scenario-based upper bound on violation probability
    beta:    float
        beta value in (0,1) defining the confidence on the epsilon level
    sieving: bool
        Discard constraints for better performance
    Samples: Array
        Array of samples to compile bounds with
    kwargs_opt: dictionary
        Arguments to be passed in optimization function
    """
    if Samples is None:
        N = CalcTools.Calc_N(Graph.n_theta,epsilon,beta)
        print("Samples drawn: {} \n".format(N))
        Samples = Graph.MultiSample(N)
    if sieving:
        res,_,eps = GraphOPF.Sieve_constraints_optimize(Graph, Samples, epsilon, beta, **kwargs_opt)
    else:
        lb,ub = GraphOPF.OPF_constraints(Graph,Samples)
        res = GraphOPF.Optimize(Graph, lb, ub, M = Samples.shape[1], **kwargs_opt)
        eps = 0
    return res,eps

def Single_run_monte_carlo(Graph,N,Samples = None,kwargs_opt={}):
    """
    Run a single OPF optimization using the Monte-Carlo approach

    Graph: Graph
        Graph used to provide constraints

    N: int
        Number of samples
    Samples: Array
        Array of samples to compile bounds with
    kwargs_opt: dictionary
        Arguments to be passed in optimization function
    """
    if Samples is None:
        print("Samples drawn: {} \n".format(N))
        Samples = Graph.MultiSample(N,1)
    _,res_list = GraphOPF.Monte_Carlo_optimize(Graph, Samples, **kwargs_opt)
    return np.average(res_list[res_list[:,1]==1,0]),1-np.average(res_list[:,1])


def Test_feasibility_and_successrate(Graph,epsilon,beta,M,maxiter=200):
    """
    Find the empirical violation probability of graph, and if neither 0 or 1, find the average successrate of sampling a collectively feasible realization for the scenario approach

    Graph: Graph
        Graph used to provide power loads and power dynamics
    epsilon: float
        epsilon value in (0,1) defining scenario-based upper bound on violation probability
    beta:    float
        beta value in (0,1) defining the confidence on the epsilon level
    M: int
        Number of time instances for which the OPF must be run in parallel

    maxiter: int
        Number of tries to find probability of sampling a feasible realization (default:200)
    """
    N = CalcTools.Calc_N(Graph.n_theta*M,epsilon,beta)
    Samples = Graph.MultiSample(N*10,M)

    res_MC,Viol_emp = Single_run_monte_carlo(Graph,N*10,Samples)

    print("Violation probability: {} ".format(Viol_emp))

    if Viol_emp == 0.0:
        print("Feasible")
        return Viol_emp,0.0
    elif Viol_emp == 1.0:
        print ("Not feasible")
        return Viol_emp,0.0
    else:
        print("Partly feasible")
```

```
    s_list = []

    for i in range(maxiter):
        if i%int(maxiter/10) == 0:
            print(str(10*i//int(maxiter/10))+"%")
        Samples = Graph.MultiSample(N,M)
        try:
            res1,eps = Single_run_scenario(Graph, sieving = False, Samples = Samples, kwargs_opt = {})
            if res1.success:
                s_list += [1]
            else:
                s_list += [0]
        except:
            s_list += [0]

    return Viol_emp,sum(s_list)/maxiter

def Find_feasible_realization(Graph,N,M,maxiter=100,kwargs_opt = {}):
    """
    Resample graph until a set of collectively feasible samples are found.

    Graph: Graph
        Graph used to provide constraints
    N: int
        Number of samples
    M: int
        Number of time instances for which the OPF must be run in parallel

    maxiter: int
        Number of tries before search for feasible realization is given up (default:100)
    kwargs_opt: dictionary
        Arguments to be passed in optimization function
    """
    for i in range(maxiter):
        Samples = Graph.MultiSample(N,M)
        try:
            res,eps = Single_run_scenario(Graph, sieving = False, Samples = Samples, kwargs_opt = kwargs_opt)
            if res.success:
                return Samples
            else:
                pass
        except:
            pass
    raise ValueError("No feasible realizations found")



def Scenario_applicability_rate(Loadfunction, Edges_lst, Samples_list = None, Samples_check = None, M = 1, N = 1, repeat = 50, N_novel = 100000, kwargs_opt = {}, FileDir = None):
    """
    Computes the scenario approach to empirical violation probability, for each modification proposed

    Loadfunction: function
        Function to load virgin graph
    Edges_lst: list
        List of proposed modifications. Structured as
            [
             [
              [list of labels]
              array of proposed modification as
                  0) begin node (index)
                  1) end node (index)
                  2) type of modification (1: addition, 0: upgrade)
                  3) Admittance value of edge
                  4) Cost of modification
                  5) Time spent upto this modification
              Graph object after optimization
             ]
            ]
    Samples_list:list
        List of samples of size repeat, for which optimal solutions are calculated and checked against new samples
    Samples_check:array
        Samples to be checked for feasibility against optimal solutions
    M: int
        Number of time instances for which the OPF must be run in parallel
    N:      int
        Size of individual sample array
    repeat: int
      The number of iterations for each sample size to find empirical violation probability
    kwargs_opt: dictionary
        Arguments to be passed in optimization function
    FileDir: str
        Directory in which results will be saved, None for no information saved (default: None)
    """
    G_reload = Loadfunction()

```

```python
    V_min = np.tile(np.array([Node.constraints[Node]["low"] for Node in G_reload.Nodes_list]),(M,1))
    V_max = np.tile(np.array([Node.constraints[Node]["high"] for Node in G_reload.Nodes_list]),(M,1))

    V_0 = 1
    V_L = 1

    Prob_array = np.zeros((Edges_lst.shape[0]+1,repeat))
    if Samples_list is None:
        Samples_list = [Find_feasible_realization(G_reload,N,M,4000,kwargs_opt = kwargs_opt) for i in range(repeat)]
    if Samples_check is None:
        Samples_check = G_reload.MultiSample(N_novel,M)

    for k in range(repeat):
        Prob = 0
        Samples = Samples_list[k]
        res,_ = Single_run_scenario(G_reload, Samples = Samples, kwargs_opt = kwargs_opt, sieving = False)

        for Sample in Samples_check:
            V = GraphOPF.Single_step(G_reload,Sample,res.x)
            if ((V>=V_min).all() and (V<=V_max).all()):
                Prob += 1

        Prob_array[0,k] = 1-Prob/(N_novel)

    if not(Edges_lst is None) and Edges_lst.shape[1] == 6:
        for j in range(Edges_lst.shape[0]):
            if Edges_lst[j,3]:
                #Add edge
                G_reload.Add_edge(G_reload.Nodes_list[Edges_lst[j,0].astype(int)],G_reload.Nodes_list[Edges_lst[j,1].astype(int)],Edges_lst[j,2])
            else:
                #Update edge
                for edge in G_reload.Edges_list:
                    begin,end = edge.connections
                    if set([G_reload.Index_Lookup[begin],G_reload.Index_Lookup[end]]) == set(Edges_lst[j,[0,1]].astype(int)):
                        break
                G_reload.Update_edge(edge, new_admittance = Edges_lst[j,2])

            G_reload.Comp_Impedance_matrices(G_reload.Slack_bus_connections)

            for k in range(repeat):
                Prob = 0
                Samples = Samples_list[k]
                res,_ = Single_run_scenario(G_reload, Samples = Samples, kwargs_opt = kwargs_opt, sieving = False)


                for Sample in Samples_check:
                    V = GraphOPF.Single_step(G_reload,Sample,res.x)
                    if ((V>=V_min).all() and (V<=V_max).all()):
                        Prob += 1

                Prob_array[j+1,k] = 1-Prob/(N_novel)
    else:
        Prob_array = Prob_array[[0],:]

    if not(Edges_lst is None) and Edges_lst.shape[1] == 6:
        plt.boxplot(np.transpose(Prob_array[:,:]),positions = range(0,Edges_lst.shape[0]+1),sym="")
    else:
        plt.boxplot(np.transpose(Prob_array),positions = [0], sym="")

    plt.xticks(range(0,Edges_lst.shape[0]+1))
    plt.xlabel("Number of modifications")
    plt.ylabel("Probability of solution being feasible for a new random sample")
    plt.title("Number of samples to find solution: "+str(N))
    if FileDir != None:
        plt.savefig(FileDir+"/Scenario_applicability_rate_box_"+str(N)+"_Samples.png")
    plt.show()

    return Prob_array




def Supporting_constraints(Loadfunction, Edges_lst, Samples, M = 1, N = 1, kwargs_opt = {}, FileDir = None):
    """
    [Not used for final report]
    Computes the number of support constraints, for each modification proposed

    Loadfunction: function
        Function to load virgin graph
    Edges_lst: list
        List of proposed modifications. Structured as
            [
             [
```

```
              [list of labels]
              array of proposed modification as
                  0) begin node (index)
                  1) end node (index)
                  2) type of modification (1: addition, 0: upgrade)
                  3) Admittance value of edge
                  4) Cost of modification
                  5) Time spent upto this modification
              Graph object after optimization
             ]
            ]
    Samples: Array
        Array of samples to compile bounds with
    M: int
        Number of time instances for which the OPF must be run in parallel
    N: int
        Number samples
    kwargs_opt: dictionary
        Arguments to be passed in optimization function
    FileDir: str
        Directory in which results will be saved, None for no information saved (default: None)
    """
    G_reload = Loadfunction()

    V_min = np.tile(np.array([Node.constraints[Node]["low"] for Node in G_reload.Nodes_list]),(M,1))
    V_max = np.tile(np.array([Node.constraints[Node]["high"] for Node in G_reload.Nodes_list]),(M,1))

    V_0 = 1
    V_L = 1

    Supp_array = np.zeros((Edges_lst.shape[0]+1))

    lb,ub = GraphOPF.OPF_constraints(G_reload,Samples, V_0, V_L)
    supp = GraphOPF.Support_constraints(G_reload, lb, ub, M = 1, **kwargs_opt)
    Supp_array[0] = len(supp)

    if not(Edges_lst is None) and Edges_lst.shape[1] == 6:
        for j in range(Edges_lst.shape[0]):
            if Edges_lst[j,3]:
                #Add edge
                G_reload.Add_edge(G_reload.Nodes_list[Edges_lst[j,0].astype(int)],G_reload.Nodes_list[Edges_lst[j,1].astype(int)],Edges_lst[j,2])
            else:
                #Update edge
                for edge in G_reload.Edges_list:
                    begin,end = edge.connections
                    if set([G_reload.Index_Lookup[begin],G_reload.Index_Lookup[end]]) == set(Edges_lst[j,[0,1]].astype(int)):
                        break
                G_reload.Update_edge(edge, new_admittance = Edges_lst[j,2])

            G_reload.Comp_Impedance_matrices(G_reload.Slack_bus_connections)

            lb,ub = GraphOPF.OPF_constraints(G_reload,Samples, V_0, V_L)
            supp = GraphOPF.Support_constraints(G_reload, lb, ub, M = 1, **kwargs_opt)
            Supp_array[j+1] = len(supp)
    else:
        Supp_array = Supp_array[[0]]

    return Supp_array

def Generate_improvement_report(Loadfunction, Edges_lst, epsilon, beta, N_MC, k_repeat, N_novel, Samples = None, kwargs_opt = {}, method = "Monte Carlo", FileDir = None):
    """
    Generate an improvement report (validation stage) of the graph and the proposed modifications

    Loadfunction: function
        Function to load virgin graph
    Edges_lst: list
        List of proposed modifications. Structured as
            [
             [
              [list of labels]
              array of proposed modification as
                  0) begin node (index)
                  1) end node (index)
                  2) type of modification (1: addition, 0: upgrade)
                  3) Admittance value of edge
                  4) Cost of modification
                  5) Time spent upto this modification
              Graph object after optimization
             ]
            ]
    epsilon: float
        epsilon value in (0,1) defining scenario-based upper bound on violation probability
    beta:    float
        beta value in (0,1) defining the confidence on the epsilon level
    N_MC:    int
        Number of samples used for Monte-Carlo part of verification stage
```

```
    k_repeat:int
        Number of repititios in finding scenario applicability rate
    N_novel: int
        Number of samples used for finding scenario applicability rate for each repitition
    Samples: Array
        Array of samples to compile results with
    kwargs_opt: dictionary
        Arguments to be passed in optimization function
    method: string
        Method used for grid expansion optimization. Either 'Scenario' or 'Monte Carlo' (default)
    FileDir: str
        Directory in which results will be saved, None for no information saved (default: None)
    """

    G_reload = Loadfunction()

    N = CalcTools.Calc_N(G_reload.n_theta,epsilon,beta)


    if Samples is None:
        print("Sampling")
        Samples = Find_feasible_realization(G_reload,N,1,2000,kwargs_opt = {}) if method == "
            Scenario" else G_reload.MultiSample(N_MC)


    report = np.zeros((Edges_lst.shape[0]+1,6))
    #Number of edges added, cost incurred, function value, violprob

    if method == "Scenario":
        res,_,eps = GraphOPF.Sieve_constraints_optimize(G_reload, Samples, epsilon, beta,**{**
            kwargs_opt,**{"ftol":10**-9}})
        report[0,2] = res.fun
        report[0,4] = eps


        for i in range(Edges_lst.shape[0]):
            report[i+1,0] = report[i,0]+1
            report[i+1,1] = report[i,1]+Edges_lst[i,4]

            if Edges_lst[i,3]:
                #Add edge
                G_reload.Add_edge(G_reload.Nodes_list[Edges_lst[i,0].astype(int)],G_reload.
                    Nodes_list[Edges_lst[i,1].astype(int)],Edges_lst[i,2])
            else:
                #Update edge
                for edge in G_reload.Edges_list:
                    begin,end = edge.connections
                    if set([G_reload.Index_Lookup[begin],G_reload.Index_Lookup[end]]) == set(
                        Edges_lst[i,[0,1]].astype(int)):
                        break
                G_reload.Update_edge(edge, new_admittance = Edges_lst[i,2])

            G_reload.Z_p, G_reload.Z_q = G_reload.Comp_Impedance_matrices(G_reload.
                Slack_bus_connections)
            try:
                pass
                res,_,eps = GraphOPF.Sieve_constraints_optimize(G_reload, Samples, epsilon, beta,
                    **{**kwargs_opt,**{"ftol":10**-9}})#[]

                report[i+1,2] = res.fun
                report[i+1,4] = eps
                report[i+1,5] = Edges_lst[i,5]

            except GraphOPF.OptimizationError:
                print("Unfeasible realization, improvement report generation terminated")
                report = report[:i+1,:]
                break

        report[:,4] = np.average(Scenario_applicability_rate(Loadfunction,Edges_lst,None,None,1,N,
            k_repeat,N_novel,kwargs_opt),axis=1)


    elif method == "Monte Carlo":
        _,res_list = GraphOPF.Monte_Carlo_optimize(G_reload, Samples, **kwargs_opt)
        report[0,2] = np.average(res_list[res_list[:,1]==1,0])
        report[0,4] = 1-np.average(res_list[:,1])
        for i in range(Edges_lst.shape[0]):
            report[i+1,0] = report[i,0]+1
            report[i+1,1] = report[i,1]+Edges_lst[i,4]

            if Edges_lst[i,3]:
                #Add edge
                G_reload.Add_edge(G_reload.Nodes_list[Edges_lst[i,0].astype(int)],G_reload.
                    Nodes_list[Edges_lst[i,1].astype(int)],Edges_lst[i,2])
            else:
                #Update edge
                for edge in G_reload.Edges_list:
                    begin,end = edge.connections
                    if set([G_reload.Index_Lookup[begin],G_reload.Index_Lookup[end]]) == set(
                        Edges_lst[i,[0,1]].astype(int)):
```

```
                        break
                G_reload.Update_edge(edge, new_admittance = Edges_lst[i,2])

            G_reload.Comp_Impedance_matrices(G_reload.Slack_bus_connections)
            _,res_list = GraphOPF.Monte_Carlo_optimize(G_reload, Samples)
            report[i+1,2] = np.average(res_list[res_list[:,1]==1,0])
            report[i+1,4] = 1-np.average(res_list[:,1])
            report[i+1,5] = Edges_lst[i,5]
    else:
        raise KeyError ("Incorrect method given")

    report[:,3] = 100*(report[0,2]-report[:,2])/report[0,2]
    if FileDir != None:
        respath = FileDir+"/Improvement_Report_"+method+".txt"
        np.savetxt(respath,report)
    return report






def PlotResult(labels, Improvement_report, figures, runs = 1, FileDir = None, shape = (6.4,4.9)):
    """
    Plots results gathered by Generate_improvement_report function

    labels: list
        List of list of labels. Structured as
            [
             [
              [list of labels]
             ]
            ]
        Structure of Edges_lst also accepted. 'list of labels' are joined and used as legend entries
    Improvement_report: list
        Individual improvement reports for all labels entries, as generated by Generate_improvement_report function. Used as datapoints
    figures: dictionary of dictionary
        Selection of which plots to draw. Structured as:
            {Plot title: {x axis title: index to find data in Improvement_report, y axis title: index to find data in Improvement_report}}
    runs:    int
        Number of simulation studies done per setting
    FileDir: str
        Directory in which results will be saved, None for no information saved (default: None)
    shape: tuple
        Shape of plots (default = (6.4, 4.9))
    """
    colors = ["tab:blue","tab:orange","tab:green","tab:red","tab:purple","tab:brown","tab:pink","tab:gray","tab:olive","tab:cyan"]

    for title in figures.keys():
        plt.figure(figsize = shape)
        plt.title(title)
        xlabel,ylabel = figures[title].keys()
        plt.xlabel(xlabel)
        plt.ylabel(ylabel)
        for i in range(int(len(labels)/runs)):
            if runs == 1:

                # [] Uncomment for log-y plot
#                 Improvement_report[i][:,figures[title][ylabel]] /= Improvement_report[i][0,figures[title][ylabel]]
#                 Improvement_report[i][:,figures[title][ylabel]] = np.log10(Improvement_report[i][:,figures[title][ylabel]])
                # []


                label = ", ".join(labels[i][0])
                label = label.replace("w_3",r'$w_4$').replace("w_2",r'$w_3$').replace("w_1",r'$w_2$').replace("w_0",r'$w_1$')

                x = Improvement_report[i][:,figures[title][xlabel]]
                y = Improvement_report[i][:,figures[title][ylabel]]
                plt.plot(x,y,"o-",label = label,color = colors[i%len(colors)])
            else:
                for run in range(runs):

                    x = Improvement_report[run][i][:,figures[title][xlabel]]
                    y = Improvement_report[run][i][:,figures[title][ylabel]]
                    plt.plot(x,y,".:",color = colors[i%len(colors)])

                N = 1000x = np.linspace(0.01,min([Improvement_report[run][i][-1,figures[title][xlabel]] for run in range(runs)]),N)[:-1]
#                                  "max" if plotting fot values until the largest cost among runs is representend (with plotting artefacts near budget)
                y = np.zeros(N-1)
                for run in range(runs):
                    for j in range(N-1):
                        if len(Improvement_report[run][i][:,figures[title][xlabel]])>1:
```

```
                            for k in range(len(Improvement_report[run][i][:,figures[title][xlabel]])-1):
                                if Improvement_report[run][i][k,figures[title][xlabel]] <= x[j] and Improvement_report[run][i][k+1,figures[title][xlabel]] >= x[j]:
                                    break

                            if Improvement_report[run][i][k+1,figures[title][xlabel]] < x[j]:
                                break
                            else:
                                y[j] += (Improvement_report[run][i][k,figures[title][ylabel]]+\
                                 (x[j]-Improvement_report[run][i][k,figures[title][xlabel]])*\
                                 (Improvement_report[run][i][k+1,figures[title][ylabel]]-Improvement_report[run][i][k,figures[title][ylabel]])/\
                                 (Improvement_report[run][i][k+1,figures[title][xlabel]]-Improvement_report[run][i][k,figures[title][xlabel]]))/\
                                (sum([1 if Improvement_report[run][i][-1,figures[title][xlabel]] >= x[j] else 0 for run in range(runs)]))

                label = ", ".join(labels[i][0])
                label = label.replace("w_3",r'$w_4$').replace("w_2",r'$w_3$').replace("w_1",r'$w_2$').replace("w_0",r'$w_1$')

                plt.plot(x,y,"-",label = label,color = colors[i%len(colors)], linewidth=2.5)


        plt.legend(bbox_to_anchor=(1, 1))
        if FileDir != None:
            plt.savefig(FileDir+"/"+title+".png",dpi=300, bbox_inches="tight")
        plt.show()
    return 1

def Verification_stage(Loadfunction, epsilon, beta, N_MC, k_repeat, N_novel, Samples, Samples_MC, res_list, kwargs_opt={}, runs = (1,1,1), folder_path=None):
    """
    Compile results and draw plots from results of grid expansion optimization

    Loadfunction: function
        Function to load virgin graph
    epsilon: float
        epsilon value in (0,1) defining scenario-based upper bound on violation probability
    beta:    float
        beta value in (0,1) defining the confidence on the epsilon level
    N_MC:    int
        Number of samples used for Monte-Carlo part of verification stage
    k_repeat:int
        Number of repititios in finding scenario applicability rate
    N_novel: int
        Number of samples used for finding scenario applicability rate for each repitition
    Samples: Array
        Array of samples to compile scenario-approach results with
    Samples_MC: Array
        Array of samples to compile Monte Carlo-approach results with
    res_list: list
        List of proposed modifications. Structured as
            [
             [
              [list of labels]
              array of proposed modification as
                  0) begin node (index)
                  1) end node (index)
                  2) type of modification (1: addition, 0: upgrade)
                  3) Admittance value of edge
                  4) Cost of modification
                  5) Time spent upto this modification
              Graph object after optimization
             ]
            ]

    kwargs_opt: dictionary
        Arguments to be passed in optimization function
    runs: tuple
        Number of passes of expansion optimization, Scenario verification and Monte-Carlo verification
    folder_path: str
        Directory in which results will be saved, None for no information saved (default: None)
    """
    Imp_rep_SC_total = []
    if runs[1] >= 1:
        for run in range(1,runs[0]+1):
            if Samples is None:
                Samp = Find_feasible_realization(Loadfunction(),CalcTools.Calc_N(Loadfunction().n_theta,epsilon,beta),1,2000,kwargs_opt = {})
            else:
                Samp = Samples[run-1]

            res,_,eps = GraphOPF.Sieve_constraints_optimize(Loadfunction(), Samp, epsilon, beta,**{**kwargs_opt,**{"ftol":10**-12}})
            Imp_rep_SC = []
            print("Scenario approach improvement reports run "+str(run)+":\n")
            if runs[1] == 1:
```

```
                for i in range(len(res_list[run-1])):
                    print(", ".join(res_list[run-1][i][0]))
                    Imp_rep = Generate_improvement_report(Loadfunction,
                                                          res_list[run-1][i][1],
                                                          epsilon,
                                                          beta,
                                                          N_MC,
                                                          k_repeat,
                                                          N_novel,
                                                          Samples = Samp,
                                                          kwargs_opt = {**kwargs_opt,**{"x_start":res.x}},
                                                          method = "Scenario",
                                                          FileDir=folder_path+"/Run "+str(run)+"/"+", ".join(res_list[run-1][i][0]) if folder_path is not None else None)
                    print("Modifations  Total cost  Performance %improvement epsilon  Time")
                    print(np.round(Imp_rep,3))
                    print()
                    Imp_rep_SC += [Imp_rep]
            else:
                for j in range(runs[1]):
                    Imp_rep = []
                    for i in range(len(res_list[run-1])):
                        Imp_rep_temp = Generate_improvement_report(Loadfunction,
                                                                   res_list[run-1][i][1],
                                                                   epsilon,
                                                                   beta,
                                                                   N_MC,
                                                                   k_repeat,
                                                                   N_novel,
                                                                   Samples = None,
                                                                   kwargs_opt = kwargs_opt,
                                                                   method = "Scenario",
                                                                   FileDir=None)
                        Imp_rep += [Imp_rep_temp]
                    print("Validation run "+str(j))
                    Imp_rep_SC += [Imp_rep]


            PlotResult(res_list[run-1],
                       Imp_rep_SC,
                       {"Cost versus scenario performance":{"Cost of added edges":1,"Operational performance (scenario)":2},
                        "Cost versus out of sample guarantees":{"Cost of added edges":1, "Violation Probability (out of sample guarantee)":4},
                        "Number of modifications versus time":{"Modifications":0,"Time (seconds)":5}},
                        FileDir = folder_path+"/Run "+str(run) if folder_path is not None else None,
                        runs = runs[1])


            Imp_rep_SC_total += [Imp_rep_SC] if runs[1] == 1 else Imp_rep_SC



        PlotResult(sum(res_list,[]),
                   Imp_rep_SC_total,
                   {"Cost versus scenario performance":{"Cost of added edges":1,"Operational performance (scenario)":2},
                    "Cost versus out of sample guarantees":{"Cost of added edges":1, "Violation Probability (out of sample guarantee)":4},
                    "Number of modifications versus time":{"Modifications":0,"Time (seconds)":5}},
                    FileDir = folder_path if folder_path is not None else None,
                    runs = runs[0]*runs[1])


    Imp_rep_MC_total = []
    if runs[2] >= 1:
        for run in range(1,runs[0]+1):
            if Samples_MC is None:
                Samples_MC = Loadfunction().MultiSample(N_MC,1)

            Imp_rep_MC = []
            print("Monte Carlo improvement reports run "+str(run)+":\n")
            if runs[2] == 1:
                for i in range(len(res_list[run-1])):
                    print(", ".join(res_list[run-1][i][0]))
                    Imp_rep = Generate_improvement_report(Loadfunction,
                                                          res_list[run-1][i][1],
                                                          epsilon,
                                                          beta,
                                                          N_MC,
                                                          k_repeat,
                                                          N_novel,
                                                          Samples = Samples_MC,
                                                          kwargs_opt = kwargs_opt,
                                                          method = "Monte Carlo",
                                                          FileDir=folder_path+"/Run "+str(run)+"/"+", ".join(res_list[run-1][i][0]) if
```

```
                                                                      folder_path is not None else None)
                    print("Modifations   Total cost  Performance %improvement violprob   Time")
                    print(np.round(Imp_rep,3))
                    print()
                    Imp_rep_MC += [Imp_rep]
            else:
                for j in range(runs[2]):
                    Imp_rep = []
                    for i in range(len(res_list[run-1])):
                        Imp_rep_temp = Generate_improvement_report(Loadfunction,
                                                                   res_list[run-1][i][1],
                                                                   epsilon,
                                                                   beta,
                                                                   N_MC,
                                                                   k_repeat,
                                                                   N_novel,
                                                                   Samples = None,
                                                                   kwargs_opt = kwargs_opt,
                                                                   method = "Monte Carlo",
                                                                   FileDir=None)
                        Imp_rep += [Imp_rep_temp]
                    print("Validation run "+str(j))
                    Imp_rep_MC += [Imp_rep]


            PlotResult(res_list[run-1],
                           Imp_rep_MC,
                           {"Cost versus Monte-Carlo performance":{"Cost of added edges":1,"Operational performance (Monte-Carlo)":2},
                       "Cost versus Violation Probability":{"Cost of added edges":1,"Violation probability (fraction of infeasible samples)":4}},
                        FileDir = folder_path+"/Run "+str(run) if folder_path is not None else None,
                        runs = runs[2])


            Imp_rep_MC_total += [Imp_rep_MC] if runs[2] == 1 else Imp_rep_MC


        PlotResult(sum(res_list,[]),
                       Imp_rep_MC_total,
                       {"Cost versus Monte-Carlo performance":{"Cost of added edges":1,"Operational performance (Monte-Carlo)":2},
                        "Cost versus Violation probability":{"Cost of added edges":1,"Violation probability (fraction of infeasible samples)":4}},
                        FileDir = folder_path if folder_path is not None else None,
                        runs = runs[0]*runs[2])


    return Imp_rep_SC_total,Imp_rep_MC_total




def Correlate(labels, imp_rep_SC, imp_rep_MC, axis_labels, runs = 5 , FileDir = None, shape = (6.4,4.9)):
    """
    Runs correlation study between two types of result

    labels: list
        List of list of labels. Structured as
            [
             [
              [list of labels]
             ]
            ]
        Structure of Edges_lst also accepted. 'list of labels' are joined and used as legend entries
    imp_rep_SC: list
        List of improvement reports as generated by Verification_stage
    imp_rep_MC: list
        List of improvement reports as generated by Verification_stage
    axis_labels: dict
        Correlation study to be run, with axis labels on as keys, lists as values.
        values ordered as [- "Monte Carlo" | "Scenario" - , - index in improvement report -]
    runs: int
        Number of simulations run per setting
    FileDir: str
        Directory in which results will be saved, None for no information saved (default: None)
    shape: tuple
        Shape of plots (default = (6.4, 4.9))
    """
    colors = ["tab:blue","tab:orange","tab:green","tab:red","tab:purple","tab:brown","tab:pink","tab:gray","tab:olive","tab:cyan"]


    plt.figure(figsize = shape)
    xlabel,ylabel = axis_labels.keys()
    title = "Correlation study of \n"+xlabel+"\n and \n"+ylabel
    plt.title(title)
    if axis_labels[xlabel][1] == 1:
        plt.xlabel(xlabel)
```

```
    else:
        plt.xlabel("Relative improvement of \n"+xlabel)
    plt.ylabel("Relative improvement of \n"+ylabel)
    Xtot = []
    Ytot = []


    for i in range(int(len(labels)/runs)):

        X = []
        Y = []
        for run in range(runs):
            print(run)
            if axis_labels[xlabel][0] == "Scenario":
                x = imp_rep_SC[run][i][:,axis_labels[xlabel][1]]
            elif axis_labels[xlabel][0] == "Monte Carlo":
                x = imp_rep_MC[run][i][:,axis_labels[xlabel][1]]

            if axis_labels[ylabel][0] == "Scenario":
                y = imp_rep_SC[run][i][:,axis_labels[ylabel][1]]
            elif axis_labels[ylabel][0] == "Monte Carlo":
                y = imp_rep_MC[run][i][:,axis_labels[ylabel][1]]

            for j in range(1,len(x)):
                if axis_labels[xlabel][1] == 1:
                    X += [(x[j]-x[j-1])]
                else:
                    X += [(x[j-1]-x[j])/abs(x[j-1])]
                Y += [(y[j-1]-y[j])/abs(y[j-1])]


        plt.plot(X,Y,"o",label = ", ".join(labels[i][0]),color = colors[i%len(colors)])
        Xtot += X
        Ytot += Y


    coef = np.polyfit(Xtot,Ytot,1)
    poly1d_fn = np.poly1d(coef)
    r_2 = np.corrcoef(Xtot,Ytot)[0,1]**2
    plt.plot([min(Xtot),max(Xtot)],poly1d_fn([min(Xtot),max(Xtot)]),":k")
    plt.text(max(Xtot)*0.8,poly1d_fn(max(Xtot)*0.7),'$r^2 = $'+str(np.round(r_2,2)),fontsize="large")


    plt.legend(bbox_to_anchor=(1, 1))

    plt.autoscale(True)
    plt.axhline(y=0, lw=2, color='k',zorder=1)
    plt.axvline(x=0, lw=2, color='k',zorder=1)

    if FileDir != None:
        plt.savefig(FileDir+"/"+title.replace("\n","")+".png",dpi=300, bbox_inches="tight")
    plt.show()


    return 1
```

GraphParameterStudy

```
import os
#os.chdir('..')

import numpy as np
import datetime
import copy

from GraphClass import *
import GraphOPF
import CalcTools
import GraphOptimizationLoop
import GraphPlot
import GraphValidation



def DirSetup(*subfolder_titles, runs = 1):
    """
    Setup directory of current run using date and time

    *subfolder_titles:
        All titles of subfolders
    """
    Master_dir = os.getcwd().replace("\\","/")+"/Runs"
    folder_path = Master_dir+'/Run_'+str(datetime.datetime.now())[:10]+\
                                "_"+str(datetime.datetime.now())[11:13]+\
                                "-"+str(datetime.datetime.now())[14:16]
    if not os.path.exists(folder_path):
```

```
        os.mkdir(folder_path)
        for run in range(1,runs+1):
            os.mkdir(folder_path+"/Run "+str(run))
            for subfolder in subfolder_titles:
                os.mkdir(folder_path+"/Run "+str(run)+"/"+subfolder)
    else:
        raise SystemError ("Duplicate directory name: folder_path")

    return folder_path



def Recursive_permutation(dictionary):
    """
    Compute all combinations of values in dictionary

    dictionary: dictionary
        dictionary of arguments and possible input values for that argument. Structured as:
            {argument name : [list of values], ...}
    """
    key,*rem = dictionary.keys()
    contents = dictionary[key]
    if isinstance(contents,list):
        pass
    else:
        contents = [contents]
    if len(rem) == 0:
        return [{key:val} for val in list(contents)]
    else:
        new_dict = {k:dictionary[k] for k in rem}
        return [{key:val , **other} for other in Recursive_permutation(new_dict) for val in list(
            contents)]



def Test_graph(Loadfunction, epsilon = 0.05, beta = 10**-5, M = 1, args_edge = {30.0}, N_MC =
    100000, k_repeat = 50, N_novel = 100000, runs = (5,1,1), SaveFig = False, args = {}, **kwargs):
    """
    Run parametric study on the grid expansion program using the combination of all arguments passed
        in args

    Loadfunction: function
        Function to load virgin graph

    epsilon: float
        epsilon value in (0,1) defining scenario-based upper bound on violation probability
    beta:    float
        beta value in (0,1) defining the confidence on the epsilon level
    M: int
        Number of time instances for which the OPF must be run in parallel
    args_edge:
        Arguments to be passed into new edge, first one to be the admittance value
    N_MC: int
        Number of samples used for Monte-Carlo validation. Monte-Carlo grid expansion uses the same
            samples as the Scenario approach as determined using epsilon and beta.
    runs: int
        How many times the optimizations and validation steps have to be ran (default = (5,1,1))
    SaveFig: bool
        Save all intermediate and final results
    args: dictionary
        Arguments used for parametric study on the grid expansion program. Structured as:
            {argument name : [list of values], ...}

    **kwargs:
        Grid expansion optimization keyword arguments to deviate from standard settings:
            "Weights" : np.array([2000,1000,10,5])
            "Cutoff" : 5
            "Budget" : 20
            "node_max" : -1
            "Leniancy" : 0.05
            "Branch_depth" : 1
            "Branch_breadth" : 1
            "kwargs_OPF" : {}
            "kwargs_opt" : {"obj_func":lambda x,C_v: C_v@x,"Cost_vector":np.ones(G.n_nodes*2)}
            "method" : "Scenario"
            "Verbose" : True
            "kwargs_plot" : {"shape":(720,720),"node_weight":6,"edge_weight":4,"edge_label":True}
        Excluded from parametric study
    """


    if len(args.keys()) >2:
        print("Warning: large number of arguments ("+str(len(args.keys()))+") may lead to long
            runtime.")



    settings = Recursive_permutation(args)
    print("Running parametric study using arguments\n"+str(settings))

```

```
    # Remove duplicate breadth value when Depth == 1
    if "Branch_depth" in args.keys() and "Branch_breadth" in args.keys():
        if 1 in args["Branch_depth"]:
            for setting in settings:
                if setting["Branch_depth"] == 1:
                    setting.pop("Branch_breadth")
            settings = [dict(t) for t in {tuple(d.items()) for d in settings}]


    folder_path = None

    if SaveFig:
        folder_path = DirSetup(*[", ".join([str(key)+" = "+str(d[key]) for key in d.keys()]) for d in settings],runs = runs[0])

    Samples_list = []
    res_list_total = []
    for run in range(1,runs[0]+1):

        G = Loadfunction()

        print("Finding feasible realization")
        N = CalcTools.Calc_N(G.n_theta*M,epsilon,beta)
        Samples = GraphValidation.Find_feasible_realization(G,N,M,2000,kwargs_opt = {})

        Samples_list += [Samples]

        print("Number of scenarios:                  {}".format(Samples.shape[0]))
        print("Number of samples:                    {}\n".format(int(np.product(Samples.shape)*G.n_delta/G.n_nodes)))

        ViolProb_array = CalcTools.Improvement_Array_generator(N,G.n_theta*M)

        kwargs_Graph_opt_base = {"Samples" : Samples,
                                 "Weights" : np.array([2000,1000,10,5]),
                                 "ViolProb_array" : ViolProb_array,
                                 "Cutoff" : 5,
                                 "Budget" : 20,
                                 "node_max" : -1,
                                 "Leniancy" : 0.05,
                                 "Branch_depth" : 1,
                                 "Branch_breadth" : 4,
                                 "kwargs_OPF" : {},
                                 "kwargs_opt" : {"obj_func":lambda x,C_v: C_v@x,"Cost_vector":np.ones(G.n_nodes*2)},
                                 "method" : "Scenario",
                                 "Verbose" : True,
                                 "kwargs_plot" : {"shape":(720,720),"node_weight":6,"edge_weight":4,"edge_label":True},
                                 "FileDir" : None}


        res_list = []
        for setting in settings:
            kwargs_Graph_opt = {**kwargs_Graph_opt_base, **kwargs}

            kwargs_temp = copy.deepcopy(setting)
            if SaveFig:
                kwargs_temp["FileDir"] = folder_path+"/Run "+str(run)+"/"+", ".join([str(key)+" = "+str(setting[key]) for key in setting.keys()])

            if "Clustering" in kwargs_temp.keys():
                kwargs_temp["node_max"] = {False:-1,True:1}[kwargs_temp.pop("Clustering")]

            Weights = kwargs_Graph_opt["Weights"]
            for i in range(kwargs_Graph_opt["Weights"].shape[0]):
                if "w_"+str(i) in kwargs_temp.keys():
                    Weights[i] = kwargs_temp.pop("w_"+str(i))
                    kwargs_temp["Weights"] = Weights

            G = Loadfunction()
            T_start = datetime.datetime.now()


            Edges_lst,Graph_upg = GraphOptimizationLoop.Optimization_loop(G,
                                                                         epsilon,
                                                                         beta,
                                                                         M,
                                                                         *args_edge,
                                                                         **{**kwargs_Graph_opt, **kwargs_temp})

            res_list += [[[str(key)+" = "+str(setting[key]) for key in setting.keys()],Edges_lst,Graph_upg]]
            T_end = datetime.datetime.now()



            print("\n")
            print("Number of modifications:           {}".format(Edges_lst.shape[0]))
```

```
            print("Total cost:                    {}".format(sum(Edges_lst[:,4]) if Edges_lst.shape[0] >= 1 else 0))
            print("")
            print("time elapsed: ")
            print(T_end-T_start)
            print("\n\n")

        res_list_total += [res_list]


    Samples_MC = G.MultiSample(N_MC,M)
    Imp_rep_SC, Imp_rep_MC = GraphValidation.Verification_stage(Loadfunction,
                                                                epsilon,
                                                                beta,
                                                                N_MC,
                                                                k_repeat,
                                                                N_novel,
                                                                Samples = Samples_list,
                                                                Samples_MC = Samples_MC,
                                                                res_list = res_list_total,
                                                                kwargs_opt = kwargs_Graph_opt_base["kwargs_opt"],
                                                                runs = runs,
                                                                folder_path = folder_path)




    return res_list_total,Imp_rep_SC,Imp_rep_MC

def LoadResults(folder_path,runs,settings = None,filename = "Edges_added.txt"):
    """
    Load results from previous parameter study

    folder_path: str
        Adress of parameter study
    runs:        bool
        Runs of parameter study
    settings:    list of dictionaries
        Settings of parameter study. If None, FindSettings will be ran
    filename:    str
        Which result to load
    """
    files = []
    if settings is None:
        settings = FindSettings(folder_path)

    for run in range(1,runs[0]+1):
        subfiles = []
        for i in range(len(settings)):
            setting = [str(key)+" = "+str(settings[i][key]) for key in settings[i].keys()]
            FileDir = folder_path+"/Run "+str(run)+"/"+", ".join(setting)
            if filename == "Edges_added.txt":
                arr = np.loadtxt(FileDir+"/"+filename,dtype=float)
                if len(arr.shape) == 1:
                    subfiles += [[setting,np.expand_dims(arr,0)]]
                else:
                    subfiles += [[setting,arr]]
            elif filename == "Samples.txt":
                arr = np.loadtxt(FileDir+"/"+filename,dtype=complex)
                if len(arr.shape) == 2:
                    subfiles += [np.expand_dims(arr,1)]
                else:
                    subfiles += [arr]
                break
            else:
                arr = np.loadtxt(FileDir+"/"+filename,dtype=float)
                if len(arr.shape) == 1:
                    subfiles += [np.expand_dims(arr,0)]
                else:
                    subfiles += [arr]
        files += [subfiles]
    return files

def FindSettings(folder_path):
    """
    Find the settings used for parameter study in folder_path

    folder_path: str
        Adress of parameter study
    """
    subfolders = os.listdir(folder_path+"\\Run 1")
    filtered = list(filter(lambda elem: '.png' not in elem,subfolders))
    perms = [[elem.split(" = ") for elem in line.split(", ")] for line in filtered ]

    permlist = []
    for perm in perms:
        dictionary = dict()
        for elem in perm:
            dictionary[elem[0]] = elem[1]
```

```
        permlist += [dictionary]

    return permlist
```

GraphPlot

```
import cv2 as cv
import numpy as np

def Draw_Edge(Frame,Pos_1,Pos_2,name,color,weight,label):
    """
    Draw edge on image

    Frame:  Array
        Image to plot edge on
    Pos_1:  tuple
        Position of first end of line
    Pos_2:  tuple
        Position of second end of line
    name:   string
        Text to be put next to line
    color:  Array
        Color of line
    weight: float
        Thickness of line
    """
    Frame = cv.line(Frame,Pos_1,Pos_2,color,weight)
    if label:
        Frame = cv.putText(Frame,name,(int((Pos_1[0]+Pos_2[0])/2),int((Pos_1[1]+Pos_2[1])/2)),cv.
            FONT_HERSHEY_SIMPLEX,weight/8,np.zeros(3,dtype=float),int(weight/2.5))
    return Frame

def Draw_Node(Frame,Pos,name,color,weight,controllable,samplable):
    """
    Draw node on image

    Frame:  Array
        Image to plot node on
    Pos:  tuple
        Position of node
    name:   string
        Text to be put next to node
    color:  Array
        Color of node
    weight: float
        Size of node
    """
    Frame = cv.circle(Frame,Pos,weight,color,-1)
    if controllable:
        Frame = cv.circle(Frame,Pos,weight,np.zeros(3,dtype=float),2)
    if samplable:
        Frame = cv.rectangle(Frame,(Pos[0]-int(weight+2),Pos[1]-int(weight+2)),(Pos[0]+int(weight+2)
            ,Pos[1]+int(weight+2)),np.zeros(3,dtype=float),1)

    Frame = cv.putText(Frame,name,(int(Pos[0]+weight*1.5),int(Pos[1]+weight*0.5)),cv.
        FONT_HERSHEY_SIMPLEX,weight/15,np.zeros(3,dtype=float),int(weight/4))

    return Frame

def Draw_Legend(shape,backgroundcolor,Types_set,weight):
    """
    Draw legend of nodes

    Shape:              tuple
        Size of image
    backgroundcolor:  Array
        Color of background in image
    Types_set:          set
        Types used in image
    weight: float
        Size of nodes in image
    """
    Word_length = weight*30
    Word_height = weight*2.5
    num_elem = len(Types_set)

    max_x = int(0.9*shape[0]/Word_length)
    max_y = int(np.ceil(num_elem/max_x))

    Frame = np.ones((int((max_y+1)*Word_height + 12),shape[1],3),dtype=float)*backgroundcolor/255
    Frame[:2,:,:] = np.zeros((2,shape[1],3))


    for j in range(max_y):
        for i in range(max_x):
            if len(Types_set)<= 0:
                break
```

```
            else:
                Pos = (int(i*Word_length+weight*1.5),int(j*Word_height+weight*1.5))

                name,colortuple = Types_set.pop()
                color = np.array(colortuple,dtype=float)
                Frame = cv.circle(Frame,Pos,weight,color,-1)
                Frame = cv.putText(Frame,name,(int(Pos[0]+weight*1.5),int(Pos[1]+weight*0.5)),cv.FONT_HERSHEY_SIMPLEX,weight/20,np.zeros(3,dtype=float),int(weight/5))


    Pos = (int(weight*1.5),int((j+1)*Word_height+weight*1.5))
    Frame = cv.circle(Frame,Pos,weight,np.zeros(3,dtype=float),2)
    Frame = cv.putText(Frame,"Controllable node",(int(Pos[0]+weight*1.5),int(Pos[1]+weight*0.5)),cv.FONT_HERSHEY_SIMPLEX,weight/20,np.zeros(3,dtype=float),int(weight/5))

    Pos = (int(Word_length+weight*1.5),int((j+1)*Word_height+weight*1.5))
    Frame = cv.rectangle(Frame,(Pos[0]-int(weight),Pos[1]-int(weight)),(Pos[0]+int(weight),Pos[1]+int(weight)),np.zeros(3,dtype=float),1)
    Frame = cv.putText(Frame,"Samplable node",(int(Pos[0]+weight*1.5),int(Pos[1]+weight*0.5)),cv.FONT_HERSHEY_SIMPLEX,weight/20,np.zeros(3,dtype=float),int(weight/5))



    return Frame



def Draw_Graph(Graph, shape = (720,720), backgroundcolor = np.array([255,255,255],dtype=float), node_weight = 10, edge_weight = 3, edge_label = True, boundary_width = 100, Legend = True):
    """
    Draw Graph and its nodes and edges

    Graph:              Graph
        Graph to be drawn

    shape:              tuple
        Size of image (default: (720,720))
    backgroundcolor:    Array
        Color of background in image (default: np.array([255,255,255],dtype=float) ; white)
    node_weight:        float
        Size of nodes in image (default: 10)
    edge_weight:        float
        Thickness of line in image (default: 3)
    boundary_width:     int
        Whitespace at edges of image (default: 100)
    Legend:             bool
        Indicates if legend should be drawn (default: True)
    """
    Frame  = np.ones(shape+(3,),dtype=float)*backgroundcolor/255

    boundary_width += node_weight

    scale_hor  = (shape[0]-boundary_width*2)/(max([*[Node.position[0] for Node in Graph.Nodes_list],1])-min([Node.position[0] for Node in Graph.Nodes_list]))
    scale_vert = (shape[1]-boundary_width*2)/(max([*[Node.position[1] for Node in Graph.Nodes_list],1])-min([Node.position[1] for Node in Graph.Nodes_list]))
    scale      = min(scale_hor,scale_vert) # Force ratio

    translation = (boundary_width-scale*min([Node.position[0] for Node in Graph.Nodes_list]), boundary_width-scale*min([Node.position[1] for Node in Graph.Nodes_list]))

    for Edge in Graph.Edges_list:
        begin,end = Edge.connections
        Pos_1 = (int(scale*begin.position[0]+translation[0]),int(scale*begin.position[1]+translation[1]))
        Pos_2 = (int(scale*end.position[0]+translation[0]),int(scale*end.position[1]+translation[1]))
        Frame = Draw_Edge(Frame,Pos_1,Pos_2,str(Edge.admittance),Edge.color/255,edge_weight,edge_label)

    Types_set = set([])
    for Node in Graph.Nodes_list:
        pos = (int(scale*Node.position[0]+translation[0]),int(scale*Node.position[1]+translation[1]))
        if Node.name == "No name":
            Frame = Draw_Node(Frame,pos,"Node {}".format(Graph.Index_Lookup[Node]),Node.color/255,node_weight,Node.controllable,Node.samplable)
        else:
            Frame = Draw_Node(Frame,pos,Node.name,Node.color/255,node_weight,Node.controllable,Node.samplable)
        Types_set.add((Node.TypeName,tuple(Node.color/255)))

    if Legend:
        Frame = cv.putText(Frame,"LEGEND",(5,shape[1]-5),cv.FONT_HERSHEY_SIMPLEX,node_weight/20,np.zeros(3,dtype=float),int(node_weight/5))
        Legend = Draw_Legend(shape,backgroundcolor,Types_set,node_weight)
        Frame = np.concatenate((Frame,Legend),axis=0)

    return Frame

```

```

def Show(Frame,title = "Grid"):
    """
    Show an image, and wait until any button is pressed to close it

    Frame: Array, list of arrays
        Image(s) to be drawn
    title: string, list of strings
        Title(s) of the window in which image will be shown
    """
    if isinstance (Frame,list):
        for i in range(len(Frame)):
            cv.imshow(title[i],Frame[i])
    else:
        cv.imshow(title,Frame)
    cv.waitKey(0)
    cv.destroyAllWindows()
    return 1
```

Some examples of code

```
#Load graph
os.chdir('ExampleGraphs')
#    [Graph script here]
from SimpleGraph_10_nodes_Gaussian        import LoadGraph
G = LoadGraph()
print("Graph imported successfully \n")


# Parameter study of optimization method
res_list,Imp_rep_SC,Imp_rep_MC = Test_graph(LoadGraph,
                                            SaveFig = True,
                                            args = {"method":["Monte Carlo","Scenario"]},
                                            runs = (5,1,1),
                                            Budget = 20)


# Parameter study for optimizing over reliability (N.B. indexing of W starts at 0)
res_list,Imp_rep_SC,Imp_rep_MC = Test_graph(LoadGraph,
                                            SaveFig = True,
                                            args = {"w_0":[2000,0]},
                                            runs = (5,1,1),
                                            Budget = 20)


# Parameter study of branch depth D
res_list,Imp_rep_SC,Imp_rep_MC = Test_graph(LoadGraph,
                                            SaveFig = True,
                                            args = {"Branch_depth":[1,2,3]},
                                            runs = (5,1,1),
                                            Budget = 20)


# Load results and run correlation study
fp = "Runs\\Run_2025-01-31_11-40"
res_list = LoadResults(fp,(5,1,1),filename="Edges_added.txt")
Imp_rep_sc = LoadResults(fp,(5,1,1),filename="Improvement_Report_Scenario.txt")
Imp_rep_mc = LoadResults(fp,(5,1,1),filename="Improvement_Report_Monte Carlo.txt")

GraphValidation.Correlate(sum(res_list,[]), Imp_rep_sc, Imp_rep_mc,
                          {"Operational performance (Monte-Carlo)":["Monte Carlo",2],
                           "Operational performance (scenario)":["Scenario",2]},
                           len(res_list),
                           FileDir = None)
```

Example of graph definition in code

```
import os
os.chdir('..')

import numpy as np
import copy

from GraphClass import *


def LoadGraph():
    num_houses = 0.25
    dist_consumer = Distribution(lambda loc,var,M,*args: np.random.normal(loc,var),1,0.3,multiplier
        =-num_houses)

    kW_p = 0.1
    dist_solar = Distribution(lambda loc,var,M,*args: np.random.normal(loc,var),1,0.3,multiplier=
        kW_p)
```

```
    A = [Supplier(name = "Node 0", position = (0, 0), constraints = {"low":0.95,"high":1.05,"ctrl_low":0,"ctrl_high":0.2}),
         Supplier(name = "Node 1", position = (0, 2), constraints = {"low":0.95,"high":1.05,"ctrl_low":0,"ctrl_high":0.2})]

    B = [Supplier(name = "Node 2", position = (3, 0), distribution = copy.deepcopy(dist_solar), constraints = {"low":0.95,"high":1.05}),
         Supplier(name = "Node 3", position = (3, 2), distribution = copy.deepcopy(dist_solar), constraints = {"low":0.95,"high":1.05}),
         Supplier(name = "Node 4", position = (1.5, 3), distribution = copy.deepcopy(dist_solar), constraints = {"low":0.95,"high":1.05})]



    C = [Network_node(name = "Node 5", position = (1.5, 2), constraints = {"low":0.95,"high":1.05}),
         Network_node(name = "Node 6", position = (1.5, 0), constraints = {"low":0.95,"high":1.05})]

    D = [Consumer(name = "Node 7", position = (1, 1), distribution = copy.deepcopy(dist_consumer), constraints = {"low":0.95,"high":1.05}),
         Consumer(name = "Node 8", position = (2, 1), distribution = copy.deepcopy(dist_consumer), constraints = {"low":0.95,"high":1.05}),
         Consumer(name = "Node 9", position = (0, 1), distribution = copy.deepcopy(dist_consumer), constraints = {"low":0.95,"high":1.05})]

    Admittance = 30.

    edges = [Edge(A[0],C[1],Admittance),
             Edge(A[1],C[0],Admittance),
             Edge(B[0],C[1],Admittance),
             Edge(B[1],C[0],Admittance),
             Edge(B[2],C[0],Admittance),
             Edge(C[0],D[0],Admittance),
             Edge(C[1],D[1],Admittance),
             Edge(C[1],D[0],Admittance),
             Edge(D[0],D[2],Admittance),
             Edge(A[1],D[2],Admittance)]

    G = Graph(A+B+C+D,edges,Slack_bus_connections = {A[0]:Admittance})

    return G
```

Bibliography

[1] T. Keyzer and M. Duintjer Tebbens, "Netbeheerders: stop met zonneparken daar waar nauwelijks vraag naar stroom is," *Nieuwsuur*, Feb 2023.

[2] N. Derbali, "Grote delen elektriciteitsnet opnieuw onder druk, geen nieuwe aansluitingen grootverbruikers," *Nieuwe Rotterdamsche Courant*, Dec 2023.

[3] R. Fares, "Renewable energy intermittency explained: Challenges, solutions, and opportunities," Feb 2024.

[4] E. Cremona and C. Rosslowe, "Grids for europe's energy transition," Mar 2024.

[5] "Kabinet komt met miljardenlening voor netbeheerder tennet," *NOS*, Jan 2024.

[6] G. L. Aschidamini, G. A. da Cruz, M. Resener, M. J. S. Ramos, L. A. Pereira, B. P. Ferraz, S. Haffner, and P. M. Pardalos, "Expansion planning of power distribution systems considering reliability: A comprehensive review," *Energies*, vol. 15, no. 6, 2022.

[7] V. N. Motta, M. F. Anjos, and M. Gendreau, "Survey of optimization models for power system operation and expansion planning with demand response," *European Journal of Operational Research*, vol. 312, no. 2, pp. 401–412, 2024.

[8] M. C. C. Giuseppe C. Calafiore, "The scenario approach to robust control design," *IEEE TRANSACTIONS ON AUTOMATIC CONTROL*, vol. 51, no. 5, 2006.

[9] M. Picallo and F. Dörfler, "Sieving out unnecessary constraints in scenario optimization with an application to power systems," *Institute of Electrical and Electronics engineers*, pp. 6100–6105, 2019.

[10] W. A. Bukhsh, C. Zhang, and P. Pinson, "An integrated multiperiod opf model with demand response and renewable generation uncertainty," *IEEE Transactions on Smart Grid*, vol. 7, no. 3, pp. 1495–1503, 2016.

[11] A. Tabandeh, A. Abdollahi, and M. Rashidinejad, "Stochastic congestion alleviation with a trade-off between demand response resources and load shedding," pp. 195–202, 2015.

[12] A. Escalera, E. D. Castronuovo, M. Prodanović, and J. Roldán-Pérez, "Reliability assessment of distribution networks with optimal coordination of distributed generation, energy storage and demand management," *Energies*, vol. 12, no. 16, 2019.

[13] N. Gong, X. Luo, and D. Chen, "Bi-level two-stage stochastic scuc for iso day-ahead scheduling considering uncertain wind power and demand response," *The Journal of Engineering*, vol. 2017, no. 13, pp. 2549–2554, 2017.

[14] J. Qiu, K. Meng, J. Zhao, and Y. Zheng, "Power network planning considering trade-off between cost, risk, and reliability," *International Transactions on Electrical Energy Systems*, vol. 27, no. 12, p. e2462, 2017. e2462 ITEES-17-0214.R1.

[15] S. Xie, Z. Hu, L. Yang, and J. Wang, "Expansion planning of active distribution system considering multiple active network managements and the optimal load-shedding direction," *International Journal of Electrical Power & Energy Systems*, vol. 115, p. 105451, 2020.

[16] J. H. Zhao, Z. Y. Dong, P. Lindsay, and K. P. Wong, "Flexible transmission expansion planning with uncertainties in an electricity market," *IEEE Transactions on Power Systems*, vol. 24, no. 1, pp. 479–488, 2009.

[17] M. Jooshaki, A. Abbaspour, M. Fotuhi-Firuzabad, G. Muñoz-Delgado, J. Contreras, M. Lehtonen, and J. M. Arroyo, "An enhanced milp model for multistage reliability-constrained distribution network expansion planning," *IEEE Transactions on Power Systems*, vol. 37, no. 1, pp. 118–131, 2022.

[18] A. K. Kazerooni and J. Mutale, "Transmission network planning under a pricebased demand response program," pp. 1–7, 2010.

[19] Z. Li, W. Wu, X. Tai, and B. Zhang, "A reliability-constrained expansion planning model for mesh distribution networks," *IEEE Transactions on Power Systems*, vol. 36, no. 2, pp. 948–960, 2021.

[20] G. Muñoz-Delgado, J. Contreras, and J. M. Arroyo, "Distribution network expansion planning with an explicit formulation for reliability assessment," *IEEE Transactions on Power Systems*, vol. 33, no. 3, pp. 2583–2596, 2018.

[21] Y. Xu, C.-C. Liu, K. P. Schneider, and D. T. Ton, "Placement of remote-controlled switches to enhance distribution system restoration capability," *IEEE Transactions on Power Systems*, vol. 31, no. 2, pp. 1139–1150, 2016.

[22] J.-H. Teng and C.-N. Lu, "Value-based distribution feeder automation planning," *International Journal of Electrical Power & Energy Systems*, vol. 28, no. 3, pp. 186–194, 2006.

[23] M. Löschenbrand, "A transmission expansion model for dynamic operation of flexible demand," *International Journal of Electrical Power & Energy Systems*, vol. 124, p. 106252, 2021.

[24] W. Tippachon and D. Rerkpreedapong, "Multiobjective optimal placement of switches and protective devices in electric power distribution systems using ant colony optimization," *Electric Power Systems Research*, vol. 79, no. 7, pp. 1171–1178, 2009.

[25] G. Levitin, S. Mazal-Tov, and D. Elmakis, "Genetic algorithm for optimal sectionalizing in radial distribution systems with alternative supply," *Electric Power Systems Research*, vol. 35, no. 3, pp. 149–155, 1995.

[26] C. Rathore and R. Roy, "Impact of wind uncertainty, plug-in-electric vehicles and demand response program on transmission network expansion planning," *International Journal of Electrical Power & Energy Systems*, vol. 75, pp. 59–73, 2016.

[27] A. Khodaei, M. Shahidehpour, L. Wu, and Z. Li, "Coordination of short-term operation constraints in multi-area expansion planning," *IEEE Transactions on Power Systems*, vol. 27, no. 4, pp. 2242–2250, 2012.

[28] Ö. Özdemir, F. D. Munoz, J. L. Ho, and B. F. Hobbs, "Economic analysis of transmission expansion planning with price-responsive demand and quadratic losses by successive lp," *IEEE Transactions on Power Systems*, vol. 31, no. 2, pp. 1096–1107, 2016.

[29] J. Wu, B. Zhang, Y. Jiang, P. Bie, and H. Li, "Chance-constrained stochastic congestion management of power systems considering uncertainty of wind power and demand side response," *International Journal of Electrical Power & Energy Systems*, vol. 107, pp. 703–714, 2019.

[30] G. Sun, J. Sun, S. Chen, and Z. Wei, "Multi-stage risk-averse operation of integrated electric power and natural gas systems," *International Journal of Electrical Power & Energy Systems*, vol. 126, p. 106614, 2021.

[31] S. Heidari and M. Fotuhi-Firuzabad, "Reliability evaluation in power distribution system planning studies," pp. 1–6, 2016.

[32] J. Qiu, "How to build an electric power transmission network considering demand side management and a risk constraint?," *Electrical Power and Energy Systems*, vol. 94, pp. 311–320, 2018.

[33] B. Barmish and P. Shcherbakov, "On avoiding vertexization of robustness problems: the approximate feasibility concept," vol. 2, pp. 1031–1036 vol.2, 2000.

[34] S. Garatti and M. C. Campi, "Risk and complexity in scenario optimization," *Mathematical Programming*, vol. 191, pp. 243 – 279, 2019.

[35] M. C. Campi and S. Garatti, "The exact feasibility of randomized solutions of uncertain convex programs," *SIAM Journal on Optimization*, vol. 19, no. 3, pp. 1211–1230, 2008.

[36] M. C. Campi and S. Garatti, "Wait-and-judge scenario optimization," *Mathematical Programming*, vol. 167, 07 2016.

[37] gridX, "Grid operators: Tso and dso explained," Jan 2024.

[38] S. Bolognani and S. Zampieri, "On the existence and linear approximation of the power flow solution in power distribution networks," *IEEE Transactions on Power Systems*, vol. 31, no. 1, pp. 163–172, 2016.

[39] M. Picallo, A. Anta, and B. De Schutter, "Stochastic optimal power flow in distribution grids under uncertainty from state estimation," pp. 3152–3158, 2018.

[40] M. Chamanbaz, F. Dabbene, and C. M. Lagoa, “Probabilistically robust ac optimal power flow,” *IEEE Transactions on Control of Network Systems*, vol. 6, no. 3, pp. 1135–1147, 2019.

[41] “Het stroomnet zit vol: hoe kan dat, en hoe erg is het?,” *RTL*, 2025.