Checking the $m - 1$ principle on large electricity distribution networks

Mathematical solution and Implementation in R

Author
Heleen Fritschy BSc

Supervisors
prof. dr. Erik Koelink
ir. Werner van Westering MSc

Radboud University Nijmegen

Alliander N.V.

April 2018

Abstract

This master thesis deals with finding an efficient mathematical algorithm to determine in what extent Alliander's medium voltage distribution networks satisfy the so-called m-1 principle. An electricity distribution network satisfies the $m-1$ principle if the network has a reconfiguration for any possible broken cable. In particular, the number of edges that may be used in a switchover for a broken cable is bounded by a positive integer k. Furthermore, a reconfiguration for a broken cable must not exceed a voltage or current capacity as computed by a load flow computation. We transformed this question of checking the $m-1$ algorithm into a mathematical problem.

We develop mathematical algorithms to check the $m-1$ principle for different values of k. They consist of different graph algorithms having a low time complexity for its purposes. More specific, as a reconfiguration of the network is a spanning tree, we use a spanning tree enumeration algorithm. In addition, we use a bridge enumeration algorithm to determine the graph-theoretically unswitchable edges. Besides the algorithms, we use two reduction methods to diminish the networks and therefore decrease the number of spanning trees, for we prove that the number of spanning trees grows very rapidly with the number of edges.

We implement the algorithms in software program R and test the implemented algorithms on Alliander's medium voltage networks. In the case of physical values chosen corresponding to a present state of the networks, our implementation took 2 hours and 8,5 hours for $k = 1$ and $k = 3$, respectively, on Alliander's entire medium voltage network of about 60.000 cables. If we accept 0,05 % of the edges to be undecided at the end of the computation, the algorithm for the case $k = 5$ takes less than a day on the entire network. Which is a considerable improvement compared to the currently used methods. Furthermore, we test a case corresponding to a future state of the network on different medium voltage networks consisting of about 100 up to 5000 edges. For those cases, the implementation of the algorithms took up to 160 seconds, 16 minutes and 10 hours for the cases $k = 1$, $k = 3$ and $k = 5$, respectively. The results demonstrate that the end product of this thesis enables checking the $m-1$ principle on large electricity distribution networks, which was previously not feasible for large networks within Alliander.

Contents

Chapter 1

Introduction

This research project is a M.Sc. Mathematics graduation project at the Radboud University in Nijmegen, in collaboration with the company Alliander. The aim of this project is to find an efficient algorithm to get insight in the reliability of Alliander's electricity networks. This algorithm is also useful to assess future energy scenarios on the electricity distribution grids.

First, this chapter gives some background information about Alliander and her challenges. After this, we describe the structure of the electricity distribution networks and hereafter the reliability principle of the studied *medium voltage* networks, called the $m-1$ principle. Finally, we formulate the main research question and provide a report overview.

1.1 Alliander

Alliander is an energy network company, meaning that she distributes electricity, gas and heat to customers, mainly households. She tries to give her three million customers access to energy every day. Alliander operates in the regions of Gelderland, Friesland, Flevoland, Noord-Holland and parts of Zuid-Holland. The production of the energy is no part of the work of Alliander. More background about Alliander can be found in her annual report [23]. In this project, we focus on the electricity distribution networks.

In the current energy transition, a lot of things change in the energy distribution networks, as described by Van Westering et al. (2016) [19]. With the use of new energy sources, like wind energy and solar energy, customers not only consume energy but produce energy too. The superfluous energy is delivered to the distribution network, which leads to a reciprocity of exchange of energy over the network. Whereas the distribution of energy was a one-way system in the past, it now requires two-way interaction. Furthermore, the new techniques operate independently from power consumption - because they depend on the weather - instead of the conventional power plants that match their power supply to the power demand.

These current developments present new challenges to the organization of the networks. The peaks of superfluous solar or wind energy could give overloads on the cables, leading to a less safe and reliable energy supply. Unfortunately, the replacement of cables is very expensive and the cables ought to stay in place for about 40 years. This creates the demand for a good prediction of the future energy use and generation by customers. Moreover, there is a need for insight in the state of the network, both now and in the future.

1.2 The structure of the electricity distribution networks

The book *Netten voor Distributie van Elektriciteit* by Van Oirsouw (2012) [16], particularly chapter 2, presents a comprehensive description of the electricity distribution networks in the Netherlands. The electricity distribution networks are split in three domains: the high voltage networks (HV), the medium voltage networks (MV) and the low voltage networks (LV). The high voltage networks transport energy over long distances, with a voltage of 110.000 to 380.000 Volt (V). The medium voltage networks operate at city scale and cover 10.000 to 20.000 Volt. The power is transported and partly distributed to big consumers. The low voltage networks distribute power at a district scale mainly to households, at a level of 400 V. The different domains are connected via transformers, which can change the voltage level. In this thesis, we only study the medium voltage networks. The MV networks of Alliander together contain about 35.000 km of underground cables. Chapter 4 covers some important information about the physics in the network.

The MV networks have some particular properties. First of all, not all cables in an MV network are used to distribute electricity to the consumers at a certain point in time. A combination of cables used is called a *configuration*. There are a lot of possible configurations that distribute the electricity to every household. The total number of cables, used or not, will be called m.

Due to the additional cables, an MV network has a meshed structure: it contains cycles. On the other hand, an actual configuration should be radial: it should not contain a cycle. In a radial network, one can detect disturbances more easily. Thus, cables from the complete network should be 'turned off' until the remaining network is radial. Furthermore, every bus (node) should be connected via a path to precisely one HV/MV transformer, this is a connection with the high voltage network. This warrants the provision of electricity to every household, but prevents dangerous situations of big voltage differences by connecting HV/MV transformers to each other.

To give an indication of the size of the total electricity distribution network of Alliander: there are about 350 HV/MV transformers, connecting the high voltage networks and the medium voltage networks, and about 45.000 MV/LV transformers, distributing energy between the MV networks and the low voltage networks. In some MV network, containing some HV/MV transformers and a lot of MV/LV transformers, there are more cables than buses. Generators and transformers are examples of buses. Altogether, there are about 60.000 cables in the entire MV network.

1.3 The $m - 1$ principle

Of course, the delivery of the electricity to the customers needs to be reliable. Alliander wants the number of outages to be minimized. The way Alliander monitors the quantity of outages is as follows: she measures the *outage consumer minutes* (OCM). This unit expresses the mean number of outage minutes per customer per year. In this way, the quantity of outages can be compared over different years.

To be able to shorten the duration of an outage, a medium voltage network should have the following property, called **the $m-1$ principle**:

> *If an outage in a cable in the MV network appears, the MV network can be switched into another configuration not using the damaged cable in such a way that every customer can be provided electricity.*

The cable with the outage could be any cable, so this $m-1$ principles guarantees a possible switch-over for every possible broken cable. By requiring the principle as a condition of an MV network, the duration of an outage can be limited to the time it takes to switch some cables on and off, instead of the whole time it takes to repair the broken cable. This principle is called the $m-1$ principle because the principle states that if we remove one cable, we could still rearrange the remaining network with $m-1$ cables in a well functioning electricity distribution network.

In addition, there are at least two requirements for the new configuration:

1. The current capacities of the cables in the configuration are not exceeded, as well as the voltage capacities of the buses.

2. Only a certain amount of cables can be switched on or off to obtain the new configuration from the original one. Say the maximum number of switched cables is k. (The switched cables do not include the outage cable.)

Alliander generally takes k to be 6. One remark about the first requirement: the current and voltage magnitudes change if one switches some cables on and off, from one configuration to another. This is why one needs to check whether the capacities are exceeded each time one tries another configuration. The current and voltage magnitudes belonging to a certain configuration can be calculated using *load flow equations*, explained in chapter 4, *Physical Background*.

One other note should be made: not every cable has a switch, although we assumed it hitherto. Logically, we could only switch cables that have a switch. Nevertheless, we can assume that the cables that are turned off in the original configuration have a switch. For if it were not, such a cable is quite useless: it is not in use now and we cannot quickly put it into service. In the sequel, we will assume these cables have a switch. Besides, the cables that are turned on not necessarily have a switch. However, we could quite easily make an opening in the network near the cable, which will function as a one-way switch: from 'on' to 'off'. Therefore, we do not have to care about the presence of a switch concerning switched-on cables.

At this point it is probably clear that it is desirable that an MV network satisfies the $m-1$ principle. However, one needs to test whether the principle applies to a certain MV network. Alliander has a software tool to do this, it is implemented in the software program *Vision*. Unfortunately, it takes a long time to run such a test. For an MV network with 136 cables, it takes 33 minutes on an ordinary laptop. To test the principle on the combination of all MV networks is almost impossible. Therefore, once a year, this big computation is done using an external server. This gives static results of the calculation, which are not actual during the year. The business management department of Alliander would probably like to have a tool to test the principle on the topical state of the network. Moreover, there is a team within Alliander, named ANDES, that develops a model to calculate the overloads of the networks in the future. It would be very useful to add the $m-1$ calculation to this model.

The challenge of this project is to establish an algorithm, using mathematical structures, that more quickly calculates whether the $m-1$ principle holds for a certain network.

1.4 Research question

At this point we can formulate the main research question as:

> *What mathematical algorithm checks the $m-1$ principle for MV networks in a reasonable time?*

The algorithm found will be implemented in the program R and will be tested on several MV networks of Alliander. In this way, ‘reasonable’ is initially defined as ‘faster than the Vision function’. The precise mathematical formulation of the research question and the detailed constraints on the configurations can be found in chapter 2, *Mathematical View*.

Note the possible difference between the mathematical definition of ‘fast’ and the actual time it takes to run an algorithm. The mathematical definition will be in terms of the *complexity*. We will study both the complexity and the execution time of the algorithms used.

1.5 Structure of this thesis

The remainder of this thesis presents a detailed description of the mathematical methods, the physical requirements, an overview of the implementation and the results of our algorithm to check the m-1 principle. Chapter 2, *Mathematical View*, formulates the main research question as a mathematical problem and proposes some potential solution strategies. Besides, it provides the relevant background in graph theory and complexities. Chapter 3, *Spanning Trees*, examines the feasibility of the chosen research strategy. It proposes two reduction methods to diminish the networks and to render the outlined strategy feasible. It also proposes a required algorithm for the strategy and proves some statements regarding its characteristics. Chapter 4, *Physical Background*, explains the necessary physics for this project. This includes the *load flow equations* that should be calculated in each configuration to compare the values with the capacities. Chapter 5, *More Graph-theoretical Tools*, continues developing the required methods for the overall algorithm to check the $m-1$ principle. After that, chapter 6, *Implementation & Overview*, displays and explains the eventual combination of the algorithms and other tools. Based on the implementation in R corresponding to the presented algorithm, chapter 7, *Results & Conclusion*, depicts the results of the algorithm applied to Alliander's entire MV network and some smaller test networks. We conclude on the performance of the algorithm afterwards. The last chapter, chapter 8, *Discussion & Future Work*, proposes some improvements of the eventual algorithm. Additionally, it generalises the solution to the $m-1$ problem to other topics and it suggests some interesting extensions of this project.

Chapter 2

Mathematical View

As formulated in chapter 1, the main research question is:

> *What mathematical algorithm checks the $m-1$ principle for MV networks in a reasonable time?*

In this chapter we translate this question into a more precise mathematical problem. Afterwards, we recall some basic principles of graph theory. Third, we approach this problem in several mathematical ways. The next chapters will refine and elaborate on the chosen approach. Last, we present the relevant background on complexities.

2.1 Mathematical problem

We abstract an MV network to a graph $G = (V, E)$. V comprises the nodes: the generators, transformers and load buses in the network. E comprises the edges: the cables between the nodes. The number of nodes is n, the number of edges is m. As the graph G represents an MV network, it has a meshed structure with a lot of cycles. We assume that the graph does not have loops (edges having identical ends), but it may have parallel edges. Therefore, the graph is not necessarily simple, for a simple graph does not have loops nor parallel edges.

To start with a basic case, we expect only one HV/MV transformer in a network. In general, there could be more HV/MV transformers, but HV/MV transformers cannot be connected to each other, which makes that case more complicated. Later on, we will study the extended case with more than one HV/MV transformer, particularly in section 5.2.

The graph G represents a complete MV network. Besides, we have the graph G^A, which represents a state of the network actually used, a configuration. Exactly the edges in G^A transport power to the consumers. The edges in $G \setminus G^A$ represent the redundant edges that are switched off and consequently have no flowing current. By the safety reasons explained in chapter 1, G^A should be a spanning tree of G, so a radial configuration of G that connects all nodes.

Given G and G^A, we need to check for every edge e in G^A which at most k edges of $G \setminus \{e\}$ need to be switched with respect to $G^A \setminus \{e\}$ to obtain a well-functioning reconfiguration G_e, if it exists. Well-functioning means for now that the reconfiguration is a spanning tree of G.

Keep in mind that we should also calculate the load flow of the new configuration G_e and compare the results with the current and voltage capacities. We will first neglect this to simplify the problem. We will append this aspect to the algorithm later on: each time we want to try a new configuration, we calculate the load flow. If at least one of the current or voltage capacities is exceeded, we delete the configuration and search for another optional solution.

The main research question is equivalent to finding a fast algorithm with the following input and output:

Wanted: efficient algorithm to check $m-1$ principle
Input: Graph G and corresponding graph G^A, a number k.
Output: For every edge e of G^A a list of at most k edges of $G \setminus \{e\}$ that need to be added to or deleted from $G^A \setminus \{e\}$ to obtain a spanning tree of G, or the sentence "no possible reconfiguration".

2.2 Background in graph theory

The graduate textbook *Graph Theory* by Bondy and Murty (2008) [1] provides a solid foundation for the terminology and principles of graph theory. The most basic definitions will not be given in this thesis, but can be found in the first chapter of Bondy and Murty (2008). Here we give some definitions and propositions relevant for the next chapters (they can be found in Bondy and Murty (2008), sections 1.1, 4.1 and 4.2). Sometimes they are just stated to avoid ambiguity between different possible definitions for one object. We do not prove the theorems and propositions, but the reader would be able to prove the theorems oneself, in all likelihood.

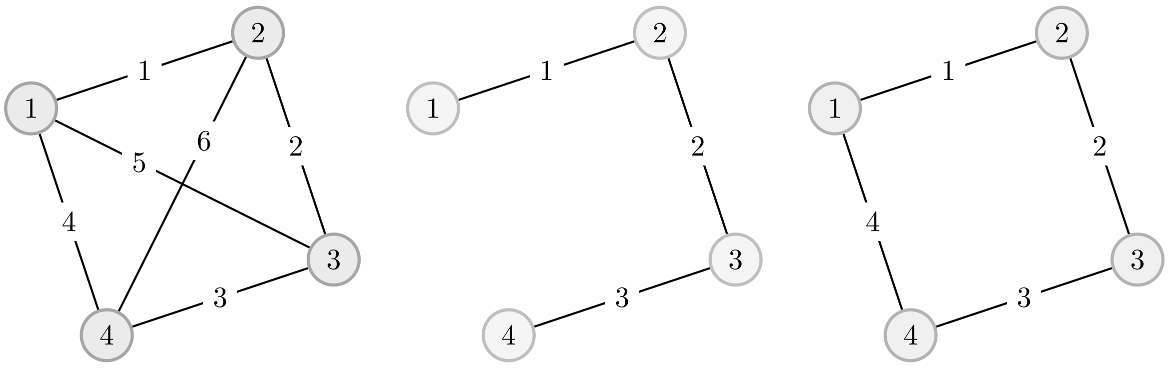
Recall the fact that we only look at graphs without loops. If we consider a graph G, it will be a such a graph, unless explicitly indicated differently. Note that the graph is not necessarily simple, for it may have parallel edges. Besides, a default assumption is that the graph is undirected. As may be expected from the context of real networks, we assume the graphs are finite.

Remark 2.1. A (simple, undirected) graph G on n nodes has at most $\binom{n}{2} = \frac{n(n-1)}{2}$ edges. A simple graph on n nodes with exactly $\binom{n}{2}$ edges is called a **complete graph** and is denoted by K_n. A subgraph (of a graph G) on n' nodes that is complete is called a n'-**clique** (of G).

Definition 2.2. A **path** is a (sub)graph whose vertices can be arranged in a linear sequence such that two vertices are adjacent if and only if (*iff*) they are consecutive in the sequence. Likewise, a **cycle** is a (sub)graph whose vertices can be arranged in a *cyclic* sequence such that two vertices are adjacent iff they are consecutive in the sequence. (A cyclic sequence means a linear sequence where the first element equals the last element.) The **length** of a path or a cycle is the number of its edges.

Remark 2.3. A path and a cycle can both be represented by a sequence of vertices (as in the definition) and by a sequence of edges. We require that in either manner vertices or edges cannot repeat in the sequence.

Example 2.4. A complete graph on four vertices, a path of length three and a cycle of length four, respectively:



Definition 2.5. A graph is **connected** if, for every partition of its vertex set into two nonempty sets V_1 and V_2, there is an edge with one end in V_1 and one end in V_2.

Proposition 2.6. *A graph is connected iff for each pair $\{v_1, v_2\}$ of its vertices there exists a path between v_1 and v_2.*

Definition 2.7. The **degree** of a vertex v in a graph G, denoted by $d(v)$, is the number of edges of G incident with v. A vertex of degree zero is called an **isolated vertex**. A vertex of degree one is called a **leaf**.

Theorem 2.8. *Let G be a graph in which all vertices have degree at least two. Then G contains a cycle.*

Definition 2.9. An **acyclic** graph is a graph that contains no cycles. A connected acyclic graph is called a **tree**.

Proposition 2.10. *Let T be a tree on $n \geq 2$ nodes. The following statements are true:*

(1) Any two vertices of T are connected by exactly one path.

(2) T has at least two leafs.

(3) The number of edges m equals $n - 1$.

Definition 2.11. A **spanning subgraph** G' of a graph G is a subgraph such that $V_{G'} = V_G$ (and $E_{G'} \subset E_G$). A **spanning tree** of a graph is a spanning subgraph that is a tree.

Theorem 2.12. *A graph is connected iff it has a spanning tree.*

For the sake of clarity, we give the next result that follows from the previous proposition and theorem:

Proposition 2.13. *Let G be a graph with n vertices. Then the following are equivalent:*

(a) G is a tree.

(b) G is connected and has $n-1$ edges.

(c) G has no cycles and has $n-1$ edges.

(d) There is a unique path in G between any two vertices.

Definition 2.14. Let G be a connected graph. An edge e in G that disconnects G upon removal, is called a **bridge** of the graph G.

Remark 2.15. Note that a bridge e of connected graph G is in every spanning tree of G. Notice also that $e = (u, v)$ is a bridge of G *iff* there is no path without e from u to v in G, by proposition 2.6.

We now gathered some basic facts about (spanning) trees. Subsequently, we present some definitions and properties about the search for a spanning tree in a given connected graph. An extended description of this so-called *tree-search* can be found in Bondy and Murty (2008) [1], section 6.1.

Definition 2.16. Let G be a graph and let X be a set of vertices of G. The set of edges of G with precisely one end in X is called the **edge cut** of G associated with X and is denoted $\partial(X)$. Note that $\partial(X) = \partial(V_G \setminus X)$. For a subgraph G' of G, we simply write $\partial(G')$ for $\partial(V_{G'})$.

Remark 2.17. Let T be a tree in a graph G. We distinguish two cases:

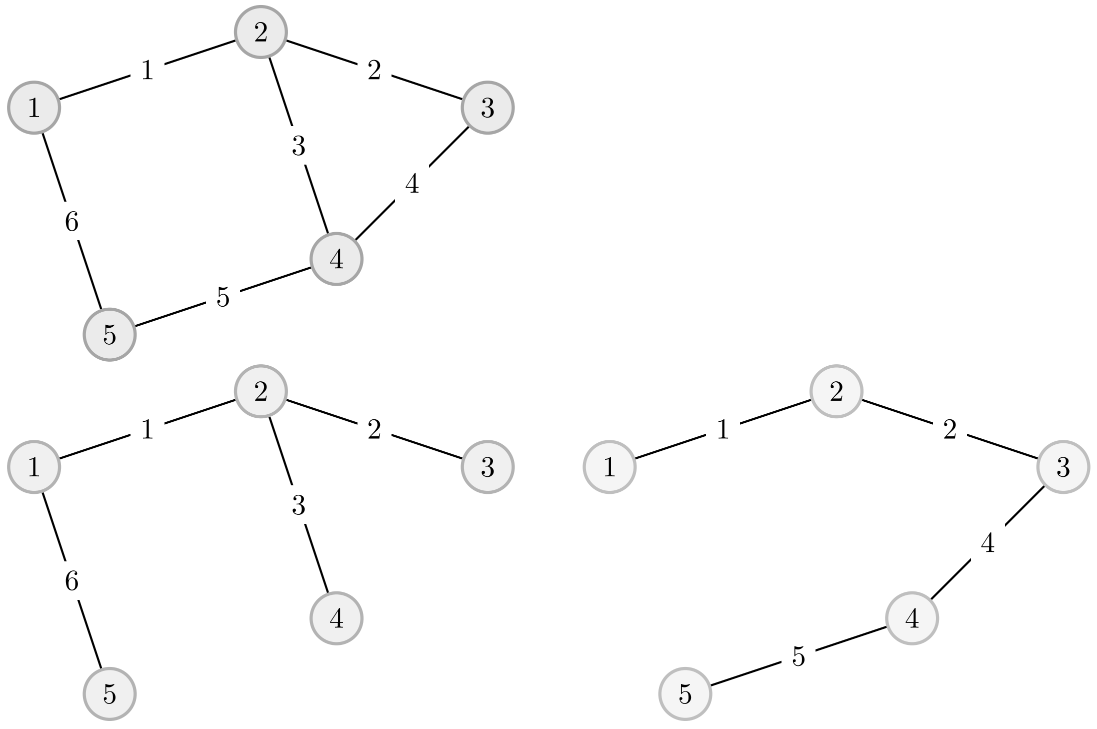
1. $V_T = V_G$: Then T is a spanning tree of G by definition.
2. $V_T \subset V_G$: We have again two possibilities:
   - (i) $\partial(T) = \emptyset$: Then there does not exist a path between any vertex in V_T and any vertex in $V_G \setminus V_T$. In that case, G is disconnected, so no spanning tree of G exists by theorem 2.12.
   - (ii) $\partial(T) \neq \emptyset$: In this case, by adding an edge $e \in \partial(T)$ to T (and hereby adding a new vertex to T), the obtained subgraph is again a tree in G. Verify this by noting that the obtained subgraph is connected (as one end of e was already in T) and does not contain cycles (as the edge e has one end that is a leaf in the extended subgraph, so cannot create a cycle).

Given this remark, we can search for a spanning tree of a graph (if it exists). Using the above distinctions, we can generate a sequence of rooted trees in G, starting with a tree consisting of a single vertex r that is the root. We are finished when either a spanning tree of the graph is found or when a tree of the graph is found whose edge cut is empty. We call this procedure a **tree-search** and the resulting tree a **search tree**.

Finally, we note that we possibly have a choice in adding a new edge (and vertex) to the tree. To determine $\partial(T)$ we need to scan the adjacency lists of the vertices already in the tree. Suppose we add a new edge to T in the order of the adjacency lists. Then the order in which we consider these adjacency lists can provide additional information on the structure of the eventual spanning tree. We will distinguish two commonly used criteria for selecting the edge and vertex to be added to the tree T. In both cases, the criterion depends on the order in which the vertices in the tree were added. Suppose for the moment that the graph is connected. Then a tree-search will give a spanning tree.

Definition 2.18. Breadth-first search is a tree-search in which the adjacency lists of the vertices of T are considered on a first-come first-served basis, i.e. they are kept in a queue where a new vertex is added to one end (the tail) and a vertex from which the current tree will be grown is taken from the other end (the head of the queue). The spanning tree returned in this way is called a **breadth-first search tree** or **BFS-tree**. **Depth-first search** is a tree-search in which the vertex added to the tree T is one which is a neighbour of as recent an addition to T as possible (a last-come first-served basis). I.e. we keep the list of vertices in a stack where a vertex is added on its top and used as next if possible or else is immediately removed. The spanning tree returned in this way is called a **depth-first search tree** or **DFS-tree**.

Example 2.19. A connected graph with associated BFS-tree and DFS-tree:



We will end with some useful definitions for describing properties of search trees in later chapters.

Definition 2.20. Let T be a tree with root r. The **level** of a vertex v in T is the length of the (unique) path between r and v. Furthermore, each vertex on that path, including the vertex v, is called an **ancestor** of v. Each vertex of which v is an ancestor is a **descendant** of v. An ancestor or descendant of a vertex is **proper** if it is not the vertex itself.

In the remainder of this thesis, we assume the given definitions and theorems are known.

2.3 Research strategies

Given the desire to find a mathematical graph algorithm to check the $m-1$ principle, one could start to search for such an algorithm. We observe that one can search for possible reconfigurations in two ways. The first and most straightforward way is to start with an edge e and search for a suitable reconfiguration without e. Subsequently, one can repeat this procedure for all other edges of G^A. The second way is to start searching for suitable reconfigurations and hereafter match possible 'broken' edges e to these reconfigurations. In other words, first search for the solutions and subsequently find broken edges for the solutions. An advantage of the second search strategy is that we might reduce the number of computations and we can find multiple solutions at once.

As a side note, we stress that we initially want to determine with 100% certainty to what extend an MV network meets the $m-1$ principle. We do not want to approximate the answer to the question by concluding for example: "With 99% certainty, the MV network meets the $m-1$ principle insofar". Consequently, we look at the problem deterministically and avoid approximation algorithms. However, in case of disappointing results for large networks one could try such algorithms too, but in this thesis we will refrain from doing so.

We propose a research strategy using spanning trees, in line with the second search strategy. Hereafter, we make a comment that we only need more than one switch in a switchover if we check the load flow in the strategy.

Strategy: Spanning trees

As a reconfiguration of G should be a spanning tree, a possible research strategy is to find all spanning trees of G and then compare a spanning tree T_j with the network G^A. If the number of edges in the symmetric difference $T_j \triangle G^A$ is greater than $k+1$, T_j cannot be constructed from G^A by switching at most k edges. (Note that we do not need to take the outage edge into account.) In that case, T_j is not a solution for any outage edge e. In the other case, the edges in $G^A \setminus T_j$ are possible outage edges for which T_j is a suitable reconfiguration. Let e be one of the edges in $G^A \setminus T_j$. The edges that need to be added or deleted to $G^A \setminus \{e\}$ to obtain T_j are precisely the edges in $(T_j \triangle G^A) \setminus \{e\}$. To conclude, by comparing each spanning tree with G^A, we can decide which edges have a possible reconfiguration. Besides, if there is a possible reconfiguration for a certain damaged edge, we find one or more lists of at most k edges that need to be switched.

The most difficult part in this research strategy is to find a list of all spanning trees of G. The next chapter, chapter 3, deals with this issue by proposing an algorithm to find all spanning trees of a graph. Before presenting the description of this algorithm, we consider the number of spanning trees of graphs (in the next chapter). This is an important aspect regarding the feasibility of applying such an algorithm to large networks. We already give away that the number of spanning trees grows very rapidly with the size of the graph. Consequently, we will try to reduce the original graph G to diminish the number of spanning trees to be considered. For example, we could only search for spanning trees that differ at most $k+1$ edges from the configuration G^A. Even though this proposal requires to extract different subgraphs of G and use the spanning tree algorithm on all these subgraphs, the total number of considered spanning trees will be much less. Additionally, we will reduce the network graphs in some other way using the so-called 'Andrei-Chicco reduction'.

Needing more switches because of the load flow computation

Here we will show: if we need more than one switch (in particular cases), then it is because of the load flow computation. In other words, if we accept a switchover immediately if it is graph-theoretically correct, then all switchovers could consist of only one switch. We prove this statement using the following lemma.

Lemma 2.21. *Let G be a connected graph and let T be a spanning tree of G. Let edge $e \in G$. Then e is a bridge in G* iff *there is no edge $d \in G \setminus \{e\}$ such that $(T \setminus \{e\}) \cup \{d\}$ is a connected graph.*

Proof. Let edge $d \in G \setminus \{e\}$. If e is a bridge in G, then e is a bridge in any spanning subgraph of G. In particular, e is a bridge in $T \cup \{d\}$. Thus, $(T \setminus \{e\}) \cup \{d\}$ is unconnected.

On the other hand, suppose there is no edge $d \in G \setminus \{e\}$ such that $(T \setminus \{e\}) \cup \{d\}$ is a connected graph. As T is a spanning tree, $T \setminus \{e\}$ consists of two components, say X_1 and X_2. By the assumption, none of the edges in $G \setminus \{e\}$ connects these two components. As a result, the ends of an edge in $G \setminus \{e\}$ either lie both in X_1 or both in X_2. Then the same holds in $G \setminus \{e\}$ itself, i.e. each edge lies either completely in X_1 or completely in X_2. As X_1 and X_2 form a partition of the vertices, the graph $G \setminus \{e\}$ is unconnected. To conclude, e is a bridge in G. □

A consequence of the lemma is that an edge e is not a bridge if and only if there is an edge d that 'repairs' T upon removal ('break down') of e. Thus, we conclude that we can either switch an edge by using one other edge as switch or we cannot switch the edge at all (in that case it is a bridge, a graph-theoretically unswitchable edge). As a result, only if we add the load flow condition, then it could be useful to look at switchovers using more switches.

We point out that if we include the load flow computation and physical condition check, then a switchover for some edge could indeed need more than one switch. A reason for this is that more switches could provide a more balanced distribution of the current over the network, whereby no voltage or current capacity is exceeded. Chapter 4 explains the details of the load flow computation. However, we deduce already that we only need more than one switch if we include the load flow computation before we accept a switchover. Consequently, the use of the maximum number of switches k is also only useful if we include the physical condition check. As the use of the spanning tree strategy presented above is dependent on k, it is also only useful if we include the physical conditions. In fact, the case $k = 1$ is much easier than the other cases and we could compute that case without using spanning trees. We present the solution to this specific case in subsection 6.3.2.

Moreover, we note here that the cases $k = 2p+1$ and $k = 2p+2$ for $p \in \mathbb{N}$ are equal. We need one more edge to close than to open, if another edge breaks down, because the initial configuration and reconfiguration of the network are both spanning trees having the same number of edges, namely $n-1$. An example: if $k = 4$, then we may switch four other edges than the broken edge. However, if we close three optional edges, we have to open precisely two edges, which in total exceeds k. If we close two optional edges, we have to open precisely one edge. Then we have used only three switches. Thus, the case $k = 4$ contains precisely the same possible switchovers as the case $k = 3$.

2.4 The speed: Time complexity

As we need the algorithm that checks the $m-1$ principle to be fast, we need some measure of the speed of an algorithm. The mathematical method to measure the speed of an algorithm is the *time complexity* of an algorithm. We can measure the required memory space of an algorithm too, using the *space complexity*. As we are interested in the speed, we will focus on the time complexity. We will give some mathematical definitions, following Sipser's book *Introduction to the Theory of Computation* (2006) [2], section 7.1.

Before we define the time complexity, we notice a couple of things. First, the time complexity is expressed in the number of steps an algorithm uses on a particular input. However, this input may depend on several parameters. In our case, the input could depend on the number of nodes n, the number of edge m and/or some other property of a given graph. Here, we simplify the time complexity of an algorithm to a function of the size of the input and do not worry about the specific shape of the input. This could be a binary string representing the input, combining all parameters. When we compute the complexity of a particular algorithm in a later chapter, we will specify it in terms of the size of the specific input if possible.

Second, we can choose between two viewpoints: considering the longest running time for inputs of a particular size, the **worst-case analysis**, or the average time for inputs of a certain size, the **average-case analysis**. We will focus on the worst-case analysis:

Definition 2.22. Let a be a deterministic algorithm that halts on all inputs. The **time complexity** of a is the function $t_a : \mathbb{N} \to \mathbb{N}$, where $t_a(x)$ is the maximum number of steps that a uses on any input of size x.

Similarly, the **space complexity** of a is the function $s_a : \mathbb{N} \to \mathbb{N}$, where $s_a(x)$ is the maximum amount of memory (in certain predefined units) that a uses on any input of size x.

Usually, we define the time complexity of a Turing machine instead of the time complexity of an algorithm. However, for our usage of the term it suffices to define it over algorithms. This saves us from an extensive and comprehensive description of Turing machines.

To avoid unnecessary determination of constants, we will use the *big-O* notation to estimate the time complexity of an algorithm:

Definition 2.23. Let functions $f, g : \mathbb{N} \to \mathbb{R}_{>0}$ be given. We say that $f(x) = O(g(x))$ if positive integers c and x_0 exists such that for every integer $x \geq x_0$: $f(x) \leq c\ g(x)$. Then we say that $g(x)$ is an **asymptotic upper bound** for $f(x)$.

Example 2.24. Let $t(x) = 5x^3 + 20x^2 + 3$.
Then $t(x) = O(x^3)$, for take $c = 6$ and $x_0 = 21$. $t(x) \neq O(x^2)$, as we could not assign c and x_0. On the other hand, $t(x) = O(x^4)$, as x^4 is larger than x^3 for $|x| > 1$. Of course, the case where g (as in the definition) is as small as possible is the most interesting case. For constant factors we simply write $O(1)$, so if $t(x) = 25$, then $t(x) = O(1)$.

As one can see, in this way we suppress constant factors. If t is a polynomial, we consider only the highest order term of t and disregard the coefficient of that term too. We will use the time complexity to measure the behaviour of an algorithm on large inputs. In such cases, the highest order term will dominate the other terms. This justifies the big-O estimation for the time complexity if it is a polynomial.

Chapter 3

Spanning Trees

Before we present an algorithm to find all spanning trees of a given graph, it is useful to know the number of spanning trees of that graph. Hereafter, we look at how the number of spanning trees of graphs increases when the number of vertices n or the number of edges m grows. As we would like to apply the final algorithm to large electricity networks, we need to be aware of the scalability of the algorithm, which is dependent of the number of spanning trees. The third section describes how the number of spanning trees can be reduced without losing possible reconfigurations. The last section provides an algorithm to find all spanning trees of a graph.

3.1 The number of spanning trees

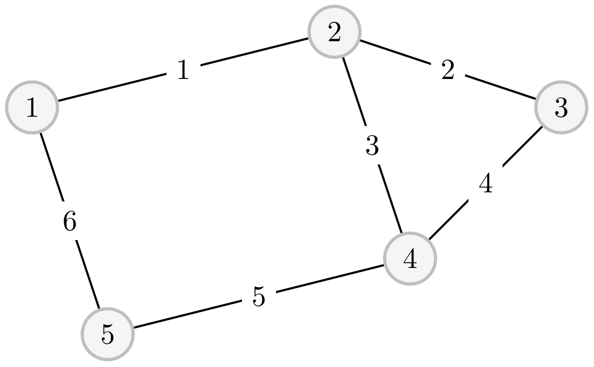
In the end of the nineteenth century, Kirchhoff contrived a way to calculate the number of spanning trees of a graph using its *incidence matrix*. Before we state Kirchhoff's theorem, also called the *Matrix tree theorem*, we need some definitions. This section follows the book *Topics in Algebraic Combinatorics* by Stanley (2013) [3], chapter 9.

Definition 3.1. Let $G = (V, E)$ be a directed graph without loops having vertices $V = \{v_1, \dots, v_n\}$ and edges $E = \{e_1, \dots, e_m\}$. Then the $n \times m$ **incidence matrix** I_G of G is defined by:

$$(I_G)_{i,j} = \begin{cases} 1 & \text{if } e_j \text{ starts in } v_i \\ -1 & \text{if } e_j \text{ ends in } v_i \\ 0 & \text{else} \end{cases}$$

In case of an undirected graph, one can arbitrarily choose which end of an edge is the starting point (and which the end point). It is probably clear that the rows of the incidence matrix represent the nodes of the graph, the columns represent the edges. Note that the sum of matrix entries along a column is zero, for each edge has one starting point and one end point. There is a one-to-one correspondence between a directed graph G and its incidence matrix I_G.

Example 3.2. An undirected graph G and a corresponding incidence matrix I_G:



$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & -1 \\ -1 & 1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 \end{pmatrix}$$

Definition 3.3. Let M be a $p \times q$ matrix. We define the $(p-1) \times q$ matrix $\widetilde{M}$ as M without its last row.

We can now state Kirchhoff's theorem, connecting the number of spanning trees of a graph with an incidence matrix corresponding to the graph.

Theorem (Kirchhoff's theorem, Matrix tree theorem)**.** *Let G be a connected graph without loops. The number of spanning trees of G equals* $\det(\widetilde{I_G} \cdot \widetilde{I_G}^T)$.

Applying Kirchhoff's theorem to example 3.2, we see that the graph contains 11 spanning trees.

At this point, the correctness of the theorem may be unclear. In the remainder of this section, we will prove Kirchhoff's theorem using some lemma's. Before that, we first explain why the choice of a direction of an undirected graph does not influence the number $\det(\widetilde{I_G} \cdot \widetilde{I_G}^T)$. Furthermore, we clarify why we could equally well choose another row to remove from a matrix M to define $\widetilde{M}$.

Although there are several incidence matrices belonging to one undirected graph, the different incidence matrices ${I_G}^i$ for a particular undirected graph G give the same result when computing $\widetilde{{I_G}^i} \cdot \widetilde{{I_G}^i}^T$. The only difference between two incidence matrices of G is the exchange of the 1 and -1 in some columns of an I_G. In the computation of $I_G \cdot {I_G}^T$ elements of a column are only multiplied with elements in the same column. By the exchange of the 1 and -1, all the individual multiplications in the computation of $I_G \cdot {I_G}^T$ give the same result, because the only elements of a column are 0's and precisely one 1 and -1. In the same way, $\widetilde{{I_G}^i} \cdot \widetilde{{I_G}^i}^T$ is the same matrix for every i, because it is only a part of the matrix $I_G \cdot {I_G}^T$. To conclude, we do not have to worry about the choice of the incidence matrix of an undirected graph: $\widetilde{I_G} \cdot \widetilde{I_G}^T$ is unique.

Second, note that $(1, \ldots, 1) \cdot I_G = 0$, so we could remove an arbitrary row from I_G without losing information. Thus, instead of defining $\widetilde{M}$ as M without its last row, we could have taken another row. As a consequence, $\det(\widetilde{I_G} \cdot \widetilde{I_G}^T)$ is not dependent on the node numbering, for we do not necessarily need to remove the row corresponding to the last node, but could take any other row, corresponding to another node.

Before we prove Kirchhoff's theorem, it is useful to get some insight in the meaning of $\widetilde{I_G} \cdot \widetilde{I_G}^T$. Therefore, we need the following definition and lemma.

Definition 3.4. Let G be a graph. The $n \times n$ **Laplacian matrix** L_G is defined by:

$$(L_G)_{i,j} = \begin{cases} d(v_i) & \text{if } i = j \\ -l & \text{if } i \neq j \text{ and there are precisely } l \text{ edges between nodes } v_i \text{ and } v_j \\ 0 & \text{else} \end{cases}$$

Clearly, L_G is symmetric.

Lemma 3.5. *Let G be a graph without loops and let an incidence matrix I_G be given. Then $L_G = I_G \cdot {I_G}^T$.*

Proof. By the definitions of matrix multiplication and the transpose, we have for nodes v_i and v_j:

$$(I_G \cdot {I_G}^T)_{i,j} = \sum_{e_k \in E_G} (I_G)_{i,k} (I_G)_{j,k} \ . \tag{3.1}$$

We consider the following cases separately:

$i = j$: If e_k is an edge with v_i one of its ends, then $(I_G)_{i,k}$ will be either 1 or -1, so $(I_G)_{i,k}(I_G)_{i,k} = 1$. If e_k does not have v_i as one of its ends, then $(I_G)_{i,k} = 0$ and $(I_G)_{i,k}(I_G)_{i,k} = 0$. By (3.1) we now see: $(I_G \cdot {I_G}^T)_{i,i} = d(v_i)$.

$i \neq j$: We solely have $(I_G)_{i,k}(I_G)_{j,k} \neq 0$ if the edge e_k has ends v_i and v_j. For $(I_G)_{i,k} \neq 0$ iff v_i is an end of e_k. In case edge e_k has ends v_i and v_j, one of $(I_G)_{i,k}$ and $(I_G)_{j,k}$ equals 1 and the other -1, so $(I_G)_{i,k}(I_G)_{j,k} = -1$. By (3.1): $(I_G \cdot {I_G}^T)_{i,j} = -l$, if there are precisely l (parallel) edges between v_i and v_j.

Hence by the definition of the Laplacian matrix, $L_G = I_G \cdot {I_G}^T$. □

Later on in this report, we will use this result about the meaning of $I_G \cdot {I_G}^T$. For now we know $\widetilde{I_G} \cdot \widetilde{I_G}^T$ equals the Laplacian matrix L_G without its last row and its last column. Currently we need the link between the number of spanning trees and determinants of certain matrices. After the following lemma and Cauchy-Binet's theorem, we arrive at the proof of Kirchhoff's theorem.

Lemma 3.6. *Let the graph G without loops be given, where G has n vertices and $n-1$ edges. If the graph G forms a spanning tree, then* $\det(\widetilde{I_G}) = \pm 1$. *If not, then* $\det(\widetilde{I_G}) = 0$.

Proof. Suppose G forms a spanning tree. Note: in the $(n-1) \times (n-1)$ matrix $\widetilde{I_G}$ the row of vertex v_n is removed with respect to I_G. Let e be an edge connected to v_n. Then the column of $\widetilde{I_G}$ indexed by e has precisely one nonzero entry, being ± 1. We form the $(n-2) \times (n-2)$ matrix $\widetilde{I_G}'$ by deleting the column and row containing this nonzero entry from $\widetilde{I_G}$. This new matrix corresponds to $\widetilde{I_{G'}}$ of the tree G' where edge e

is contracted (e is removed and the ends of e are merged into a single vertex v'_{n-1}). We notice that $\det(\widetilde{I_G}) = \pm \det(\widetilde{I_G}')$. By induction on the number of vertices n, we have $\det(\widetilde{I_{G'}}) = \pm 1$. (The case $n = 1$ is trivial, this concerns the determinant of the empty matrix.) To conclude, $\det(\widetilde{I_G}) = \pm 1$.

Now suppose G does not form a spanning tree. Then G contains a cycle. Let $C \subset E_G$ be the set of edges that forms one such cycle. Let C be represented by the cyclic edge sequence $(b_1, \ldots, b_t)$. Without loss of generality, we can modify G into a directed graph where C is a directed cycle. Then each vertex in C is just as often a starting point as an end point in C. Let $I_G[C]$ be I_G limited to the columns indexed by $b_1, \ldots, b_t$. Then each vertex in C has as many 1's as -1's in its row in $I_G[C]$. Thus, if we add the columns indexed by $b_1, \ldots, b_t$, we get the 0-column. Hence the columns of I_G are linearly dependent, so are the columns of $\widetilde{I_G}$. To end, $\det(\widetilde{I_G}) = 0$. □

Theorem 3.7 (Cauchy-Binet's theorem)**.** *Let A be an $p \times q$ matrix, let B be an $q \times p$ matrix. If $p > q$, then* $\det(A \cdot B) = 0$*. If $p \leq q$, then*

$$\det(A \cdot B) = \sum_{\substack{S \subset \{1,\ldots,q\} \\ |S| = p}} \det(A[S]) \cdot \det(B\{S\}) \ ,$$

where $A[S]$ abbreviates: matrix A limited to the columns *indexed by the elements in S and $B\{S\}$ abbreviates: matrix B limited to the* rows *indexed by the elements in S.*

We will not prove Cauchy-Binet's theorem, but a proof can be found in chapter 6 of the book *Introduction to Linear Algebra* by Marcus and Minc (1965) [4]. Now we will prove Kirchhoff's theorem:

Theorem 3.8 (Kirchhoff's theorem, Matrix tree theorem)**.** *Let G be a connected graph without loops. The number of spanning trees of G equals* $\det(\widetilde{I_G} \cdot \widetilde{I_G}^T)$*.*

Proof. Note that $\widetilde{I_G}$ is an $(n-1) \times m$ matrix. $n - 1 \leq m$, for G is connected. By Cauchy-Binet's theorem (theorem 3.7), we have:

$$\det(\widetilde{I_G} \cdot \widetilde{I_G}^T) = \sum_{\substack{S \subset \{e_1,\ldots,e_m\} \\ |S| = n-1}} \det(\widetilde{I_G}[S]) \cdot \det(\widetilde{I_G}^T\{S\}) \ .$$

In general, $M^T\{S\} = M[S]^T$, so the last equation becomes:

$$\det(\widetilde{I_G} \cdot \widetilde{I_G}^T) = \sum_{\substack{S \subset \{e_1,\ldots,e_m\} \\ |S| = n-1}} \left(\det(\widetilde{I_G}[S])\right)^2 .$$

According to lemma 3.6, if S forms the set of edges of a spanning tree of G, then $\det(\widetilde{I_G}[S]) = \pm 1$. In the other case, $\det(\widetilde{I_G}[S]) = 0$. Therefore, $(\det(\widetilde{I_G}[S]))^2 = 1$ if S forms a spanning tree, and is 0 otherwise. Hence $\det(\widetilde{I_G} \cdot \widetilde{I_G}^T)$ denotes the number of spanning trees of G. □

Kirchhoff's theorem gives a fast and easy method to calculate the number of spanning trees of a graph. We have implemented this method in R. It is a straightforward implementation using matrices. In the implementation, the graph is represented by its edges and the edges are given by their endpoints.

Note that we do not yet know which specific spanning trees a particular graph contains, we only know the number of spanning trees of that graph. Section 3.4 presents an algorithm to enumerate all spanning trees of a particular graph.

The number of spanning trees depends on the incidence matrix of the graph, as Kirchhoff's theorem shows. As a result, the number of spanning trees of a graph is dependent on the structure of the graph, meaning the specific image of the graph. Only knowing the number of nodes and the number of edges is not enough to determine the number of spanning trees.

Although the method following from Kirchhoff's theorem is very efficient to compute the number of spanning trees of a specific graph, it does not give insight in the number of spanning trees of graphs in general. One could wonder how the number of spanning trees depends on the kind of graph and whether the number of spanning trees grows exponentially when n and m grow. These issues are covered in the next section.

3.2 The growth of the number of spanning trees

This section focuses on the growth of the number of spanning trees in general. First, we give a lower bound for the number of spanning trees of any simple connected graph with a certain amount of nodes and edges. Second, we estimate the growth of the number of spanning trees as a graph grows in its number of nodes or edges.

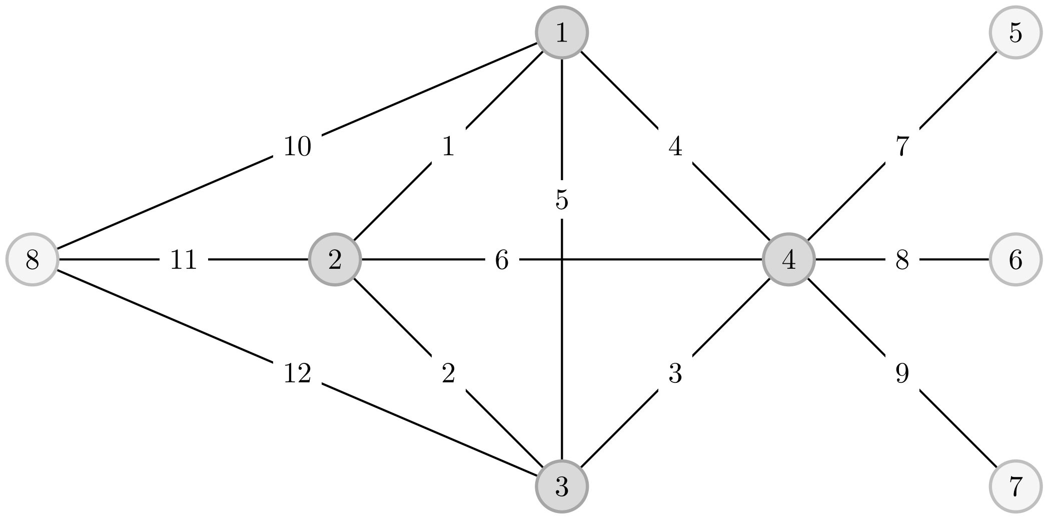
Bogdanowicz (2009) [5] proves Boesch's conjecture, providing a simple connected graph that has the least number of spanning trees of all simple connected graphs on the same number of nodes n and number of edges m. The simple connected graph having the least number of spanning trees, given n and m, is graph $L_{n,m}$:

Definition 3.9. Let n, m be given natural numbers and $n-1 \leq m$. Let k be the least positive integer such that $m \geq \frac{1}{2}(n-k)(n-k-1)+k := z$. Define $L_{n,m}$ to be a graph consisting of an $(n-k)$-clique, joined to $k-1$ vertices of degree one and one other vertex joined to $m-z+1$ vertices of the clique.

Note that $k = n-1$ always satisfies the inequality $m \geq \frac{1}{2}(n-k)(n-k-1)+k$, because $n-1 \leq m$. As a result, $n-k$ is always positive and the $(n-k)$-clique is well-defined.

We note that several graphs meet the definition of $L_{n,m}$ (for fixed n and fixed m). However, those graphs all have the same number of spanning trees, so we can treat them as if they were just one graph for convenience.

Example 3.10. Let $n = 8$ and $m = 12$, then $k = 4$, $z = 10$. Then a graph $L_{8,12}$ is:



The subgraph consisting of the four dark nodes forms the $(n-k)$-clique. Nodes $5, 6$ and 7 form the $k-1$ nodes of degree one. Node 8 is the vertex joined to $m-z+1$ nodes of the clique.

Theorem 3.11 (Boesch's conjecture)**.** *Let n, m be positive integers such that there exists a simple connected graph on n nodes and m edges. Then any simple connected graph G with n nodes and m edges has at least the number of spanning trees $L_{n,m}$ has.*

The proof of Boesch's conjecture is very technical, but can be found in Bogdanowicz (2009) [5].

Now we know which simple connected graph has the least number of spanning trees, given n and m. We still need to calculate the number of spanning trees that $L_{n,m}$ has. Fortunately, we know the number of spanning trees of a complete graph:

Theorem 3.12 (Cayley's theorem)**.** *The number of spanning trees of K_n equals n^{n-2}.*

There are a lot of ways to prove this theorem, discovered by Cayley in 1889. One could show it using the eigenvalues of the Laplacian matrix. To get more sense for the theorem, we give the combinatorial proof by Prüfer, as presented in the last appendix in Stanley (2013) [3].

Proof. We will define a map f : {spanning trees of K_n} $\to \{1, \ldots, n\}^{n-2}$. If S is a spanning tree of K_n, then the sequence $f(S) = (a_1, \ldots, a_{n-2})$ is called the *Prüfer code* of S. Afterwards, we prove that f is a bijection.

Assume that the nodes are numbered from 1 to n. Let S be a spanning tree of K_n. Let v be the *leaf* with the largest node number of S. Then there is a unique edge $e = \{v, w\}$. Define $a_1 = w$ and delete v (and therewith e) from S, giving a spanning tree on $n-1$ nodes. We continue this procedure to find $a_2, \ldots, a_{n-2}$. In the end, there remain only two vertices and one edge of S. This map is well-defined: a spanning tree always has a leaf and taking the leaf with the largest node number determines a_i uniquely.

Now we prove this map is a bijection. Let $s = (a_1, \ldots, a_{n-2}) \in \{1, \ldots, n\}^{n-2}$ be given. We try to construct a spanning tree T such that $f(T) = s$ by reasoning back from the sequence s. The first vertex to be removed from T should be the vertex with the largest node number of K_n missing from s. For if a vertex is in the sequence, it is connected to another vertex in a subgraph of T in an intermediate step, so it cannot be removed in the first step. Suppose the largest vertex of K_n missing from s, called v, is not deleted in the first step. Then this vertex cannot be a leaf (otherwise it was taken), so the vertex has at least degree two. Then a neighbour of v is removed in an intermediate step and v is put on the sequence s, else we would not end with only one edge. This cannot be the case, so the first deleted vertex is the largest vertex missing from s. Now we know the edge $\{v, a_1\}$ is part of tree T. We can continue this argument on the remainder of the sequence $(a_2, \ldots, a_{n-2})$ and the remainder of the graph $K_n - \{v\}$, in a similar way. We can repeat this reasoning $n - 2$ times and we find $n - 2$ edges of T. In the end, there are only two vertices not removed, they form the last edge.

Note: there will always be a largest vertex not in the sequence, as the sequence contains at most $n-2$ different elements and there are n different vertices. In each step, we remove one element from the sequence and one vertex from the graph. To conclude, given a sequence $s = (a_1, \ldots, a_{n-2}) \in \{1, \ldots, n\}^{n-2}$, we find at least one spanning tree T such that $s = f(T)$. Moreover, we find exactly one spanning tree T, as a spanning tree is determined by its $n - 1$ edges and we find $n - 1$ edges that must be part of the tree.

To sum up, each sequence belongs to precisely one spanning tree. Therefore, the map f is injective and surjective. Given this bijection, the number of spanning trees of K_n equals the number of elements of $\{1, \ldots, n\}^{n-2}$. The latter equals n^{n-2}, thereby completing the proof. $\square$

Combining the last two theorems, we derive the following result:

Corollary 3.13. *Let $n, m \in \mathbb{N}$ and $n - 1 \leq m$. Define: $k = \left\lceil n - \frac{3}{2} - \sqrt{\frac{9}{4} - 2n + 2m} \right\rceil$. If there exists a simple connected graph on n nodes and m edges, then every such graph has at least $(n-k)^{n-k-2} = \left(n - \left\lceil n - \frac{3}{2} - \sqrt{\frac{9}{4} - 2n + 2m} \right\rceil\right)^{n-2-\left\lceil n-\frac{3}{2}-\sqrt{\frac{9}{4}-2n+2m} \right\rceil}$ spanning trees.*

As one can see, we now have a direct formula for the minimum number of spanning trees of a simple connected graph, given only n and m.

Proof. Let natural numbers n and m be given such that a simple connected graph on n nodes and m edges exists. Using the abc-formula and doing some straightforward calculations, one can transform the statement:

$$\text{“ Let } k \text{ be the least integer such that } m \geq \tfrac{1}{2}(n-k)(n-k-1) + k \text{ ”}$$

$$\text{into: “ } k = \left\lceil n - \tfrac{3}{2} - \sqrt{\tfrac{9}{4} - 2n + 2m} \right\rceil \text{ ”.}$$

Using theorem 3.11 we know $L_{n,m}$ has the least number of spanning trees of all simple connected graphs on n nodes and m edges. We know $L_{n,m}$ has an $(n-k)$-clique, so

$L_{n,m}$ contains at least $(n-k)^{n-k-2}$ spanning trees, using theorem 3.12. (Note that $L_{n,m}$ could have many more spanning trees, but this depends on the number $z := \frac{1}{2}(n-k)(n-k-1)+k$: what degree the last node of $L_{n,m}$ needs to get. Since this degree equals 1 in some cases, we keep it general and only count the number of spanning trees of the $(n-k)$-clique.) □

We conclude from the corollary that the minimum number of spanning trees in simple connected graphs equals roughly $f(n,m) = \sqrt{-2n+2m}^{\sqrt{-2n+2m}}$. Let $-2n+2m$ grow linearly, then the function f grows eventually slower than any exponential function, as the exponent grows less than linearly (note that the base grows too, but the linear growth of the exponent is essential to have an exponential function). Of course, the function f eventually grows faster than all polynomial functions. Concluding, f cannot be approximated by a polynomial function nor an exponential function, but lies in between those two categories. f eventually exceeds any root function in logarithmic scale, as such a function has a fixed base and f has not. We call such a root function in logarithmic scale a *root exponential* function.

We now give some results with respect to the growth of the minimum number of spanning trees of a simple connected graph, depending on n and m:

1. For fixed n, the minimum number of spanning trees grows at least root exponentially if m increases.

2. For fixed m, the minimum number of spanning trees grows at least root exponentially if n decreases.

3. The minimum number of spanning trees grows at least root exponentially if m increases more than n.

For the sake of completeness, we can be more precise about the exact number of spanning trees of $L_{n,m}$ and, consequently, about the genuine least number of spanning trees of any simple connected graph on n nodes and m edges. However, since those formulas are more complex, it is harder to reason about the growth of the number of spanning trees at a glance. The conclusions derived above still hold.

Theorem 3.14. *Let K_n^w be the graph consisting of an n-clique combined with one additional vertex $n+1$ of degree w connected to vertices $n-w+1,\dots,n$. Then the number of spanning trees of K_n^w equals $w\cdot(n+1)^{w-1}\cdot n^{n-w-1}$.*

Proof. (Sketch) Similar to the proof of Cayley's theorem, one can define a (somewhat more complicated) *Prüfer code* for spanning trees of K_n^w. In this case, we define the map f : $\{\text{spanning trees of } K_n^w\} \to \{1,\dots,w\} \times \{1,\dots,n+1\}^{w-1} \times \{1,\dots,n\}^{n-w-1}$. The map f is defined in a similar way as in the proof of Cayley's theorem: one takes the leaf v with the largest node number and writes down the other end u of the edge ending in v. Then we remove vertex v and the corresponding edge and repeat this procedure recursively (until just one edge remains).

A difference with the proof of Cayley's theorem is the place in the sequence $s = (a, b_1, \dots, b_{w-1}, c_1, \dots, c_{n-w-1})$ where we write down u. If the largest leaf equals $n+1$, we define $a = u$. If the largest leaf is in the set $\{n-w+1, \dots, n\}$, we define $b_i = u$ where i is the least number such that b_i is not yet defined. If already all b_j's are defined, we define $c_l = u$ where l is the least number such that c_l is not yet defined. Lastly, if the largest leaf is in the set $\{1, \dots, n-w\}$, we define $c_l = u$ where l is the least number such that c_l is not yet defined.

To prove that f is well-defined, we first need to show that it cannot happen that there is no place for u in the sequence, i.e. it cannot be the case that we should have either $a = u$, $b_i = u$ or $c_j = u$, but we defined all a's, b_i's or c_j's already, respectively. We need to distinguish the three cases $a = u$, $b_i = u$ and $c_j = u$ to prove this. Second, we need to show that $u \in \{1, \dots, w\}$, $u \in \{1, \dots, n+1\}$ or $u \in \{1, \dots, n\}$ if it is allocated to an a, b_i or c_j, respectively. For this we need to distinguish the cases $v \in \{1, \dots, n-w\}$, $v \in \{n-w+1, \dots, n\}$ and $v = n+1$. After we have done so, we have proved that f is well-defined.

To prove that f is a bijection, we take an approach similar to the proof of Cayley's theorem (theorem 3.12), i.e. we use the same method to prove that f is a bijection. Particularly, we take a sequence $s = (a, b_1, \dots, b_{w-1}, c_1, \dots, c_{n-w-1}) \in \{1, \dots, w\} \times \{1, \dots, n+1\}^{w-1} \times \{1, \dots, n\}^{n-w-1}$ and try to construct a spanning tree T such that $f(T) = s$ by reasoning back from the sequence s. Ultimately, we will see that each sequence belongs to precisely one spanning tree. Thus, f is a bijection and the number of spanning trees of K_n^w equals the number of elements of $\{1, \dots, w\} \times \{1, \dots, n+1\}^{w-1} \times \{1, \dots, n\}^{n-w-1}$. The latter equals $w \cdot (n+1)^{w-1} \cdot n^{n-w-1}$, thereby completing the proof. □

The number of spanning trees of an $L_{n,m}$ purely depends on the $(n-k)$-clique in combination with the additional vertex joined to $m-z+1$ vertices of the clique, because the other edges are edges in branches, which are bridges. Every spanning tree contains the bridges, whereby these edges cannot increase the number of spanning trees of the entire graph. To conclude, the number of spanning trees of an $L_{n,m}$ is precisely the number of spanning trees of K_{n-k}^{m-z+1}. Combining the previous theorem and this last note, we have the following corollary:

Corollary 3.15. *The following two observations are correct:*

1. *The number of spanning trees of $L_{n,m}$ equals $w \cdot (n-k+1)^{w-1} \cdot (n-k)^{n-k-w-1}$, where we can view k and w either as:*

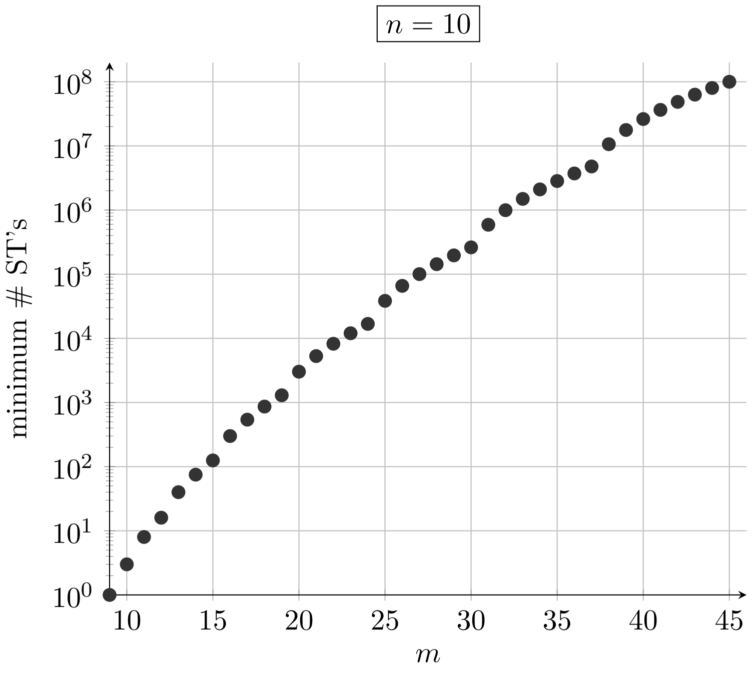
   (i) k the least integer s.t. $m \geq \frac{1}{2}(n-k)(n-k-1) + k := z$ and $w := m - z + 1$, or:

   (ii) $k := \left\lceil n - \frac{3}{2} - \sqrt{\frac{9}{4} - 2n + 2m} \right\rceil$ and $w := m + 1 - \frac{1}{2}(n-k)(n-k-1) - k$.

2. *The number of spanning trees of any simple connected graph on n vertices and m edges equals at least $w \cdot (n-k+1)^{w-1} \cdot (n-k)^{n-k-w-1}$.*

We now see that the number of spanning trees of the graph $L_{8,12}$ in example 3.10 equals $w \cdot (n-k+1)^{w-1} \cdot (n-k)^{n-k-w-1} = 3 \cdot 5^2 \cdot 4^0 = 75$. In particular, any simple connected graph having 8 vertices and 12 edges contains at least 75 spanning trees.

Example 3.16. Let $n = 10$. The plot displays the minimum number of spanning trees of a simple connected graph with 10 nodes and m edges:



The figure shows the at least root exponential growth of the minimum number of spanning trees for fixed n and increasing m. One can see that the number of spanning trees of a graph with 10 nodes and 30 edges already exceeds 100.000.

Last, note that we do not assume that the graphs corresponding to the MV networks are simple. However, in general, they have only a few parallel edges (and no loops by assumption), wherefore the minimum number of spanning trees of simple connected graphs approximates the minimum number of spanning trees of the MV network graphs. As a consequence, the number of spanning trees of the MV network graphs grows very fast for fixed n and increasing m. Therefore, we need to reduce the number of spanning trees of these graphs.

3.3 Reducing the number of spanning trees

The previous section showed that the minimum number of spanning trees is growing very rapidly with the number of edges for a fixed number of nodes. Therefore, in this section we will try to reduce the number of considered spanning trees for a given cyclic network graph. We do this by reducing the original graph G. We try to remove edges and nodes that we do not need to find all essential different spanning trees that are possible reconfigurations. After we have done so, we can search for the remaining spanning trees using a spanning tree enumeration algorithm, which is presented in the next section (section 3.4). After listing the spanning trees, we can convert the solutions back to the original graph.

We will reduce the original graph in two ways. The first and most simple way, called the k *reduction*, is based on the fact that only k switches are allowed. The second manner is called the *Andrei-Chicco reduction* and is based on the ideas of Andrei and Chicco (2008) [6] to reduce an electricity network graph without losing important information. We first describe the k reduction and afterwards the Andrei-Chicco reduction. We explain how we implemented the combination of the two reduction methods and other algorithms in chapter 6, in particular in section 6.2.

3.3.1 k reduction

Let r be the number of edges of $G \setminus G^A$. r represents the number of residual edges: the edges that are not used in the present configuration G^A but could be used in a reconfiguration. We also call the residual edges optional edges, as well as open edges in some cases. We now look at the maximum number of edges that could be closed (or opened).

Given a damaged edge e, we allow at most k edges to be switched. Note that we do not count e as one of the switches. The same number of edges should be opened and closed with respect to G^A, for both G^A and the reconfiguration should be spanning trees with $n-1$ edges. The edge e is an opened edge, but does not count as one of the k switches. To conclude, if we close $c \leq \lceil \frac{k}{2} \rceil$ edges of the r residual edges, we have to open $c-1$ edges of the original configuration G^A. If so, we have switched at most $\lceil \frac{k}{2} \rceil + \lceil \frac{k}{2} \rceil - 1 \leq k$ edges in total, which is allowed.

As we require at most $\lceil \frac{k}{2} \rceil$ edges from the r residual edges to be closed, most likely a lot of all spanning trees of G are not permitted. Those spanning trees would have more than $\lceil \frac{k}{2} \rceil$ edges from the r residual edges closed (assuming $\lceil \frac{k}{2} \rceil < r$). To avoid computing such ineffective spanning trees, we could reduce the graph G to a subgraph of it having only $\lceil \frac{k}{2} \rceil$ residual edges. Then all of the spanning trees of the subgraph are permissible reconfigurations (ignoring still the load flow condition). Unfortunately, there could be a lot of such subgraphs, but in most cases the sum of the numbers of all spanning trees of all these subgraphs will still be less than the number of spanning trees of G. This is because the minimum number of spanning trees grows root exponentially with one additional edge. Furthermore, we may assume that k is relatively small ($k = 6$ in Alliander's case) and r is substantially larger than k, which is advantageous for the k reduction method.

To sum up, the k reduction procedure consists of splitting graph G in $\binom{r}{\lceil \frac{k}{2} \rceil}$ graphs consisting of G^A added with $\lceil \frac{k}{2} \rceil$ residual edges of G. For convenience, we call the consecutive execution of (a) the selection of $\lceil \frac{k}{2} \rceil$ residual edges (to extend the initial spanning tree), (b) the enumeration of all spanning trees in this graph and (c) the deduction of possible switchovers for certain edges, a 'k reduction' itself. In other words, we mean by a k reduction: the computation of spanning trees and the determination of switchovers (including the load flow check) of one k reduced graph. One can see this as one iteration of the total k reduction procedure.

Although the k reduction procedure ensures that we limit our attention to acceptable spanning trees (reconfigurations) and avoid the computation of a lot of unacceptable reconfigurations, the number of k reductions is in some cases still too many to perform all within an acceptable time period. For example, if we take the western part of Alliander's MV network of about 25.300 cables, 22.500 nodes and $r = 2800$, then the number of k reductions equals (if $k = 6$): $\binom{r}{\lceil \frac{k}{2} \rceil} = \binom{2800}{3} = 3.654.747.600$. This is a quite extreme case, but even if a k reduction would only take 0,001 seconds, then the total time to check the $m-1$ principle for this network would take over 42 days. For the sake of clarity, we would then have completely computed whether each edge in the western region has a switchover or not, including physical conditions. We will explain how the k reduction method still is useful, but present a solution to the problem of too many k reductions in Alliander's case afterwards.

Usefulness

First of all, this k reduction method could work for smaller MV networks. For example, if we take an MV area of about 1800 cables, 1600 nodes and $r = 200$ (still $k = 6$), then $\binom{r}{\lceil \frac{k}{2} \rceil} = \binom{200}{3} = 1.313.400$. This could be a manageable number if the computations within a k reduction are quite fast. In that case, we could complete all k reduction computations within a couple of hours. At that point, we know a switchover for each switchable edge and can conclude that all remaining undecided edges after all k reductions are not switchable, for we have tried all possible reconfigurations before the termination of all k reductions.

In addition, this method also works for larger MV networks for which all graph-theoretically switchable edges have at least one acceptable reconfiguration (so they have a switchover). For in this case, we do not need to execute all k reductions, because we can stop if we have determined a switchover for all theoretically switchable edges. We can determine all graph-theoretically unswitchable edges beforehand, these are the *bridges* of the graph. Section 5.1 explains how to find all bridges.

If we shuffle the possible combinations of $\lceil \frac{k}{2} \rceil$ remaining edges to add to the initial configuration beforehand too, we would most likely find all needed switchovers long before the execution of all k reductions. We implemented this randomized k reduction procedure in R. It indeed functioned fast for Alliander's entire MV network, provided that all edges that graph-theoretically have a switchover actually have a switchover (accepted by the load flow). However, we are also interested in edges that have a switchover graph-theoretically, but in practice turn out not to have a switchover that is accepted by the load flow. We need another good idea for this case, which is presented below.

Improvement

We would like to check the $m-1$ principle for Alliander's entire MV network at once, but as we look at the entire network, a lot of the combinations of $\lceil \frac{k}{2} \rceil$ remaining edges are not that interesting. For example, in all likelihood a combination of three remaining edges of which one in Amsterdam, one in Arnhem and one in Den Helder will be equally informative as the addition of each of the three remaining edges one at a time. For the remaining edges are too far apart to provide switchover information using multiple remaining edges at once. (An edge in Amsterdam cannot be switched by using an edge in Den Helder or Arnhem.) To conclude, a large part of the k reduction combinations is not useful.

As a result, we would like to limit the k reductions to the (possible) useful combinations. We achieve this by using certain information about 'substation areas'. A substation area (Dutch: OS-gebied) is a part of the MV network consisting of just one HV/MV transformer and associated nodes and edges. The required information is which substation areas are interrelated at the MV level and which are not. Then we can take all combinations (of $\lceil \frac{k}{2} \rceil$ remaining edges) within each substation area and all combinations within neighbouring substation areas, but we miss the unnecessary combinations in areas far apart.

We go deeper into what we call 'neighbouring' substation areas. Suppose we have a list of all MV links between substation areas, each MV link given as a specific MV edge that connects two specific substation areas (having certain names). These MV links are always optional edges, for different HV/MV tranformers may not be connected to each other. Suppose that such an edge that forms a link between two substation areas is included in both substation areas. The remainder of this subsection considers the interesting cases for Alliander, these are the cases that we add one, two or three optional edges in a k reduction. In the case that we add only one optional edge, we just take each optional edge one by one in the k reduction procedure.

Take therefore the case that we add two optional edges in each k reduction. If we do not include an MV link as at least one of the two optional edges, then it only makes sense to take combinations of optional edges inside one substation area. We determine these combinations for all substation areas anyway. On the other hand, if we do include an MV link as one of the two optional edges, it makes sense to combine that edge with any edge in either the first or the second substation area that edge belongs too (this could be an edge that is also connected to a third area). However, as the MV links lie in both substation areas, we have already included these combinations after we took all combinations of optional edges within one substation area. To conclude, in the case we add two optional edges in a k reduction, it suffices to take all combinations of optional edges within substation areas.

Now we look at the case that we add three optional edges. In the first place, we take all combinations of three optional edges within each substation area, comparable to the previous case. In addition to this, we also take the combinations of one MV link, one optional edge in the first substation area the link belongs to and one optional edge in the second substation area the link belongs to. These are suitable combinations of three optional edges. Additionally, we show that we then have formed all suitable combinations of three optional edges.

We consider the most extreme but suitable case, where we combine three optional edges that are all MV links and combine four different substation areas. Say optional edge 1 connects areas A and B, optional edge 2 areas B and C, optional edge 3 areas C and D. In the first instance, it seems like we should combine four substation areas to include this combination. However, as MV links belong to both substation areas, we did include this combination of areas yet. For if we take the combinations in substation areas B and C, we include the combination of these optional edges.

The other suitable combinations of three optional edges belong to at most three substation areas. If we consider combinations in precisely three substation areas, then we need at least two MV links to connect these areas (so to make the combination suitable). The two MV links have a substation area in common, so for the three optional edges together we need only two substation area names. Thus, we found such a combination already. Concluding, we find all suitable combinations by taking all combinations of three optional edges within each substation area and taking all combinations of three optional edges in two substation areas, of which one optional edge is a connection between the areas.

Using this selection of only suitable combinations of $\lceil \frac{k}{2} \rceil$ optional edges, the number of k reductions of the western part of Alliander's MV network equals 2.164.671 (if $k = 6$), which is much more feasible. Chapter 6, particularly subsection 6.3.1, presents an overview of all elements of the eventual algorithm to check the $m - 1$ principle. The k reduction that is part of the scheme includes this extension of the k reduction notion. The data table OSGRAPH, described in subsection 6.1.2, allows for this improvement of the k reduction in the implementation in R.

3.3.2 Andrei-Chicco reduction

Andrei and Chicco (2008) [6] describe a realistic reduction of an electricity distribution network to find all radial configurations (the spanning trees). They assume the same properties of an electricity distribution system as we described in chapter 1 and 2. We will use some of their findings to develop our own reduction algorithm. Based on their ideas, we divide the process for finding the relevant spanning trees into the following steps:

1) Create a reduced network. The number of vertices and edges will be reduced according to the application of three key principles described below. In addition to the reduction, we store some information about the reduction. We could also find switchover solutions for some of the edges of G^A using the reduction only.

2) Identify all spanning trees by applying a spanning tree enumeration algorithm (free to choose) to the reduced network. As already mentioned, the next section (section 3.4) presents such an algorithm.

3) Convert the spanning trees found in step 2) to 'general' spanning trees of the original network. By 'general' we mean the structure of some comparable spanning trees of the original graph. If required, one could distinguish the different spanning trees of one general structure using the reduction information.

Note that we would apply a spanning tree enumeration algorithm as in step 2) anyway, but now we would apply it to the reduced network. Step 1) executes the reduction itself, hereafter called the *Andrei-Chicco reduction*. This is what we will describe in more detail in this subsection. As we apply the spanning tree enumeration to the reduced network instead of the original network, we need step 3) to transform the results on the reduced network to the original network. We will deal with the details of this transformation at the end of this subsection and return to it in chapter 6 (sections 6.2 and 6.3).

The key principles of the Andrei-Chicco reduction

As announced, in step 1) we need some procedures to actually reduce an original network graph. Eventually, we save the reduced graph and a list of the original edges subdivided into *categories*, groups of similar edges. We allocate a certain integer to each category. Using the categories and the reduced graph, we could largely recover the original graph. (We save the original graph too, so this is not necessary, but it is useful to have certain structures of the graph together quickly.) These are the three key principles of the Andrei-Chicco reduction to reduce a graph:

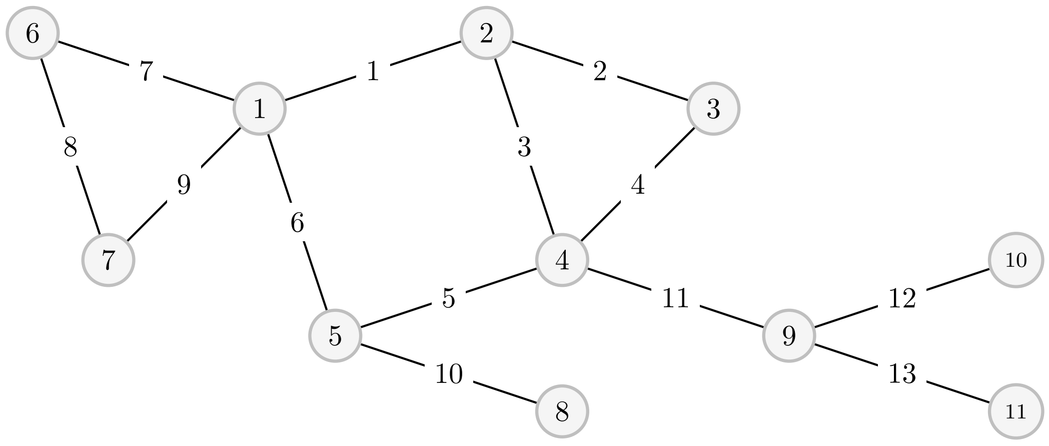
1. Suppose there is a leaf v in the graph G. If the edge e connected to v breaks down, then v can never be powered without using e. Therefore, we can conclude that e has "no possible reconfiguation". Besides, we remove e from G, because we know e will be part of every spanning tree of G but has no effect on the rest of the tree. Furthermore, every edge in the branch (the string/strand) ending in v has no possible reconfiguration, as a defect of the edge will isolate v too. Consequently, we also note that the edges in the branch have "no possible reconfiguration" and we delete them from G. All edges in branches get category zero.

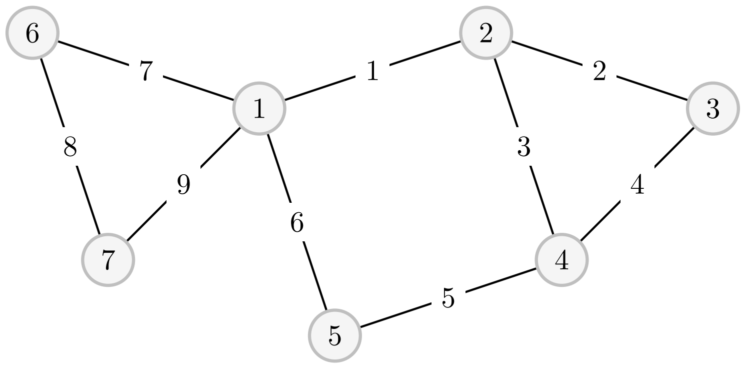
2. Let p be a path in G where all intermediate vertices have degree two, but the starting point and end point of p have degree greater than two. (By 'path' we mean such a specific path hereafter.) Then we know that either all edges in p will be in use in a spanning tree of G or precisely one edge of p will be opened, for otherwise a part of the path would be isolated. Thus, there are only two real different options for p in every spanning tree. Therefore, we can regard the path p as just one edge in G from the starting point to the end point of p. To be clear, we remove all intermediate vertices and all edges of p and add one edge from start to end of p, the *path edge*. On top, we remember all removed edges by storing them

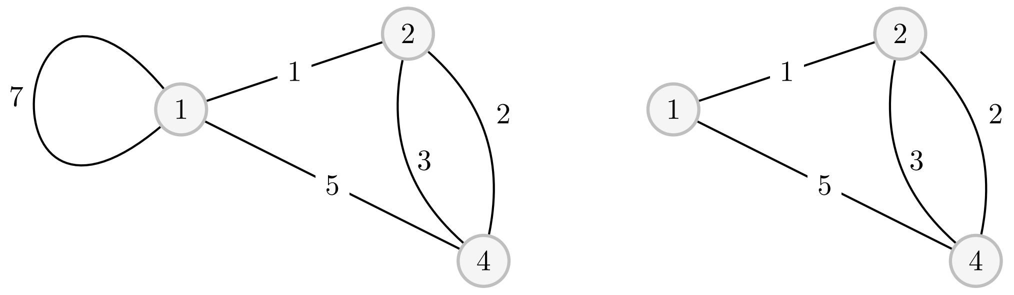
as additional information of the reduction. We give the removed edges of one path all the same category, a certain positive integer. (We allocate different categories to different paths.) In this way, we know in the end which edges form a deleted path. Given a spanning tree of the reduced graph, we can find a spanning tree of G by using all edges of the path if the path edge is used in the reduced spanning tree. In the other case where the path edge is not in use in the reduced spanning tree, we open precisely one edge of the path.

3. After executing principle 2, we possibly have created a loop (an edge with identical ends). A spanning tree of the reduced graph will never contain this loop and as a consequence the corresponding path in the original graph has precisely one opened edge. Which edge in the path will be opened can be chosen freely. Therefore, we can remove the loop from the reduced graph. Besides, we give the original path a special category, namely the initial category with a minus sign. We know that if an edge of the original path (which forms the loop) breaks down, we can reconfigure the network by closing the edge in the path that is opened.

Example 3.17. The sequential execution of the three key principles on an exemplary graph G:







Comparing our reduction principles with the reduction of Andrei and Chicco (2008) [6], there are two main differences. First, Andrei and Chicco (2008) assume several supply points to the network, being HV/MV transformers in our case. We assume only one HV/MV transformer per network primarily. However, we treat the case of multiple HV/MV transformers too, but only in section 5.2. Second, Andrei and Chicco (2008) do not mention the case of forming a loop separately, but we thought this is useful to reduce the graph even further.

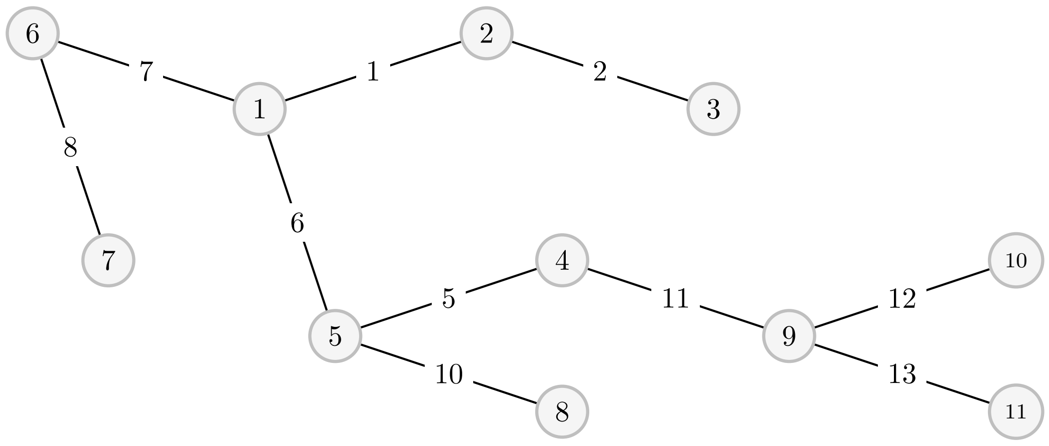
We notice three things about the key reduction principles. First of all, we note that we reduce the graph in the order of the key principles: we first remove the branches, afterwards remove the vertices of degree two (in certain paths) and finally remove arisen loops. If we would change the order, the reduced graph would possibly change too. For example, if we would apply key principle 1 later on, there could arise a vertex of degree two that will not be removed as we already executed principle 2. Moreover, we have to add that the resulting graph after the sequential execution could still be reduced according to one of the three principles. We come back to this at the very end of this section.

Second, we remark that the Andrei-Chicco reduced graph is not necessarily a simple graph, even if the original graph G is. (We could perform the Andrei-Chicco reduction on any graph, but in practice only execute it on a graph without loops.) An example: G can have two internally disjoint paths with the same starting point and end point of degree greater than two, where all intermediate vertices have degree two. In the reduction, both paths will be replaced by a single edge, giving multiple edges between two points. Fortunately, this does not matter for the spanning tree enumeration algorithm. There are of course no loops in the reduction by the third principle.

Last, as the three principles either remove leafs, nodes of degree two or loops (and some nodes of degree ≥ 3 associated with edges in branches, we will come back to that), the principles have no effect on graphs without loops with only vertices of degree greater than two, because there are no edges in branches in that case. Thus, any graph without loops with minimum degree greater than two reduces in no way after the execution of the principles. In particular, any complete graph where $n \geq 4$ remains the same after the 'reduction'.

This sounds as if the Andrei-Chicco reduction is not so suitable, for the number of spanning trees of graphs with only vertices of degree greater than two is especially large, because these graphs have relatively many edges. However, we should not forget which kind of graphs we are studying: Alliander's MV networks. In practice, these network graphs turn out to have a lot of nodes of degree two. Therefore, the Andrei-Chicco reduction is still a good option to decrease the number of spanning trees. Andrei and Chicco (2008) designed their algorithm especially for electricity distribution networks to find spanning trees. Subsection 6.2.2 comes back to the specific reduction factor resulting from this reduction on Alliander's networks.

Suppose the actual state of the network G^A of the exemplary graph G in example 3.17 is:



In other words, edges 3, 4 and 9 are the residual (optional) edges of the network. Using Kirchhoff's theorem (theorem 3.8), we see that the number of spanning trees of the original graph G equals 33, whilst the number of spanning trees of the Andrei-Chicco reduced graph equals only 5. Thus, in this example, the reduction lowered the number of spanning trees substantially.

The additional reduction information associated with the Andrei-Chicco reduction of G, called the *categories*, is:

Category	**Edges**
0	10, 11, 12, 13
1	1
2	2, 4
3	3
4	5, 6
−5	7, 8, 9

Note that all edges in the same path in the original graph G that is reduced to one edge in the reduced graph get the same category. Edges in different paths in the original graph get different categories. In the first place, all such paths get a positive number as category. If the resulting edge in the reduced graph is a loop, then the corresponding edges in the original graph get a negative category (the initial category with a minus sign).

Further note that each edge in a branch gets category zero. This is always the case for edges in branches, even if there are multiple branches or branches that intersect (as in the example). It is okay to give all edges in branches the same category, for we do not need more specific information about them. This is because the edges in branches are unswitchable and this is immediately enough as 'switchover information' (actually 'no switchover information'). As all four edges 10, 11, 12 and 13 in the example are edges in branches, we see that branches could contain intermediate vertices having degree ≥ 3 (here vertex 9).

In our implementation in R, we remove the edges in branches in the following manner (and we save these edges by assigning them to category zero): we first remove all edges having an end of degree one (a leaf). These edges are definitely edges in branches. We move on by deleting all edges having a leaf as end in the *intermediate* graph, i.e. the resulting graph after one such move. We keep on iterating this procedure until the resulting graph does not contain any leaf. In the example above, we would first remove edges 10, 12 and 13, and hereafter remove edge 11 in the next iteration. After that, the graph does not contain any leaf anymore.

We will briefly explain why this method correctly removes all edges in branches. First, we note that this method *only* removes edges in branches. An edge not in a branch will not get a leaf as end, because each end is incident with at least one other edge not ending in a branch (verify that it is in a branch otherwise). Then both ends of the non-branch retain degree ≥ 2 (with the edge itself included). Consequently, the procedure will not remove the non-branch.

On the other hand, if an edge is in a branch, then at least one end is only incident with branches. By recursion on the edges incident with that end (not the edge itself), we conclude that we eventually delete all edges incident with that end with this procedure. At that time, the end is a leaf, so we delete the edge itself. Thus, this method removes precisely all edges in branches.

Deriving switchover information from the Andrei-Chicco reduction

For the sake of clarity, note that the two outputs of the Andrei-Chicco reduction are the reduced graph (vertices and edges) and the list of original edges subdivided into categories. We can derive switchover information from these outputs without using spanning trees, if we combine these with the edges in the original configuration (or equally well with the list of original optional edges). For each of the three key principles we can determine certain switchover information already:

1. As mentioned before, all edges in branches are unswitchable. Therefore, we immediately save the 'switchover information' of these edges by concluding "no switch

possible". We could do this immediately if we detect edges in branches or later on by using the category zero. (In our implementation, we first completely Andrei-Chicco reduce the graph and draw switchover conclusions in the main algorithm afterwards.)

2. Here we confine ourselves to paths that do not become a loop during the reduction (we take a closer look at the loops afterwards). As explained in key principle 2, at most one edge of a path could be open, for a part of the path would be isolated otherwise. Consequently, there is at most one optional edge in each path. Using the list of edges in the initial spanning tree, we know which edges are optional, namely the rest of the edges. We find each path with associated path edge by taking each occurring category that is a positive integer one by one. Take such a path (in the original graph) and associated path edge e (in the reduced graph). Using the list of optional edges in the original graph, we can easily check whether one of the optional edges occurs in the path. If this is not the case, then all edges in the path are in use and we cannot directly determine switchover information. If, however, there is an optional edge q in the path (so exactly one), then we can switch all other edges in the path by using the optional edge. We can immediately save this switchover information, so listing for each edge in the path except q that a switchover of the edge consist of just q. (We still need to check the load flow, as in chapter 4.)

   Verify that we could indeed switch some edge e in the path with optional edge q, for the difference is that the edges between e and q are now connected to the rest of the graph via the edge q instead of via the edge e. This does not disconnect a part of the graph, as we may assume the initial graph was connected. Furthermore, this change cannot form a cycle.

3. Now we look at the paths that become a loop during the reduction. In such case the starting point equals the end point of the path, so the path is a cycle. Still there is at most one optional edge in the path. However, there is at least one optional edge too, for the path is a cycle and the configuration does not contain cycles. Take a path and associated loop edge e by taking a negative integer occurring as category. Then there is precisely one optional edge q, so we can switch all other edges in the path by using the optional edge, for the same reason as above. We save this switchover information as we did for the other paths.

Using these rules, we find the following switchover information for the graph in example 3.17:

Edge	Switches if possible	Key principle
2	4	2
7	9	3
8	9	3
10	no switch possible	1
11	no switch possible	1
12	no switch possible	1
13	no switch possible	1

We see that we obtain quite a lot of switchover information already using the Andrei-Chicco reduction, without the use of new spanning trees. However, remember that we could still reject these switchovers by the physical conditions.

The only undecided edges in the example are edges 1, 5 and 6, as we only need to decide switchovers for the edges of G^A and not for the optional edges (edges 3, 4 and 9). We need one or more spanning trees of the Andrei-Chicco reduced graph to find switchovers for edges 1, 5 and 6. Section 2.3 explained this extraction of switchover information from spanning trees globally. Below, we will describe this method in combination with the Andrei-Chicco reduction, which belongs to step 3) in the beginning of this section. We assume we executed step 2) correctly, although we only describe how to do this in the next section (section 3.4).

Using spanning trees in the Andrei-Chicco reduction

As shown in section 2.3, we can find switchovers by comparing the initial spanning tree with a new spanning tree found. In particular, we look at the symmetric difference between the spanning trees. We first describe some consequences of the Andrei-Chicco reduction on the initial spanning tree. After that, we present the logical rules for finding switchovers using spanning trees in the Andrei-Chicco reduction.

We present the logical rule to find the initial spanning tree in the Andrei-Chicco reduced graph (ACR graph). We indicate this graph by G_{ACR} and the initial spanning tree in the ACR graph by G^A_{ACR}. Let an original network graph G, a corresponding initial spanning tree G^A and the corresponding Andrei-Chicco reduced graph G_{ACR} be given. We derive the initial spanning tree G^A_{ACR} corresponding to G_{ACR} in the following way: a path is completely in the initial spanning tree G^A *iff* the path edge is in the ACR spanning tree G^A_{ACR}. We explain why this is a wise choice to transform the initial spanning tree.

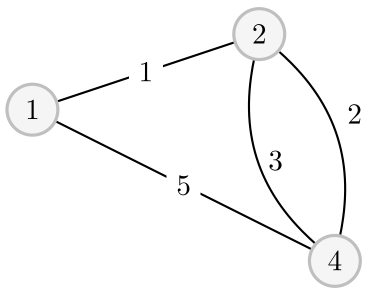
If a path is completely in the initial spanning tree G^A, then all edges in the path are closed. Therefore, it makes sense to close the path edge, i.e. the path edge is present in the ACR spanning tree G^A_{ACR} .

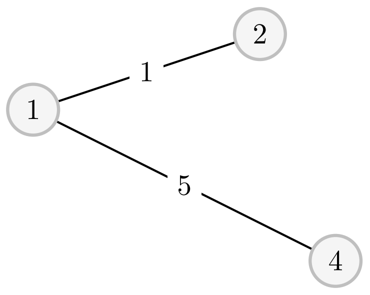
On the other hand, if a path is not completely in the initial spanning tree G^A, then there is precisely one optional edge in the path (as we saw before). The starting and end point of the path must apparently not be connected via the path, for there was no optional edge in the path otherwise. If we would close the optional edge in the path, then there is a cycle in the initial 'spanning tree' G^A, for the spanning tree contains too many edges at that moment. The Andrei-Chicco reduced graph G_{ACR} preserves most of the structures of the original graph (except for the branches and loops). Therefore, we should still not connect the starting and end point of the path via the reduced path in G_{ACR}, to make sure G^A_{ACR} really is a spanning tree in G_{ACR}. As the reduced path is just the path edge, this path edge should be open in G_{ACR}. To conclude, if the path is not completely in the initial spanning tree G^A, then the path edge is not in the ACR spanning tree G^A_{ACR}.

Although this definition of the initial ACR spanning tree G^A_{ACR} is a logical definition in a sense, it has a bit of a strange effect on the optional edges in G_{ACR}: an edge e is optional in G_{ACR} *iff* only one of the edges in the corresponding path in the original graph G (of which e is the path edge) is optional.

For by definition, the optional edges in G_{ACR} are the edges in $G_{\text{ACR}} \setminus G^A_{\text{ACR}}$. However, by the definition of G^A_{ACR}, edge e is optional in G_{ACR} *iff* the corresponding path is not completely in the initial spanning tree G^A. The latter means that it has precisely one optional edge among the edges in the path. Thus, we have to take into account that an optional edge in G_{ACR} does not lead to only optional edges in the associated path in G.

In the example of this subsection, we have the following Andrei-Chicco reduced graph G_{ACR} and corresponding initial ACR spanning tree G^A_{ACR}, respectively:





Note that G^A_{ACR} is indeed a spanning tree. However, it is somewhat confusing that the edges 2 and 3 in G_{ACR} are optional edges, but the edges 2 and 3 in G are not necessarily optional edges. In fact, edge 2 in G is not an optional edge. The only thing we know is that precisely one edge of the corresponding path in G is optional. In the example, edge 4 is an optional edge in the path consisting of edges 2 and 4 in G.

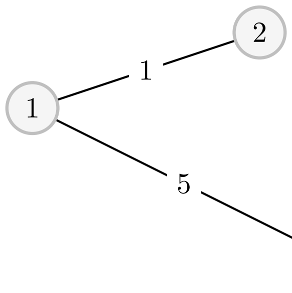
We now describe some consequences of the Andrei-Chicco reduction for the determination of switchovers using new spanning trees found. We first recapitulate the old approach without the Andrei-Chicco reduction. Let T_j be a new spanning tree found (which differs at most $k+1$ edges from G^A, we may assume this by the application of the k reduction). Section 2.3 explained that we can switch edges in $G^A \setminus T_j$ using the symmetric difference $T_j \triangle G^A$. In particular, we can switch edge $e \in G^A \setminus T_j$ by turning on or off precisely all edges in $(T_j \triangle G^A) \setminus \{e\}$. The number of edges in $(T_j \triangle G^A) \setminus \{e\}$ is equal to or less than k, which is precisely accepted.

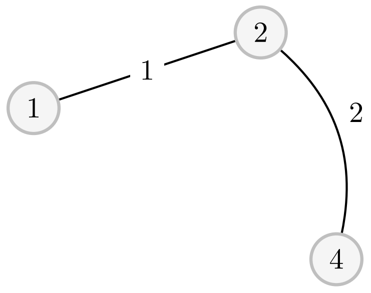
Incorporating the Andrei-Chicco reduction in this strategy needs some care, because the edges in the Andrei-Chicco reduction represent whole chains of edges (the paths) in the original graph. Let T_j now be a new spanning tree of G_{ACR} and let path edge $e \in G^A_{\text{ACR}} \setminus T_j$. Then e is open in the graph corresponding to T_j. Similar to the definition of G^A_{ACR}, we note that *precisely* one original edge in the path corresponding to e is open in the reconfigured original graph. On the other hand, e is present in G^A_{ACR} and thus all original edges in the path were closed in the original graph. Within this path of closed edges, we have now to open precisely one edge. Consequently, we can possibly find switchovers for all original edges in the path corresponding to e, as we have a choice of which edge to open in this path.

We have a similar choice for some of the path edges in $(T_j \triangle G^A_{\text{ACR}}) \setminus \{e\}$. We first present the case where we have no choice. If $d \in (T_j \triangle G^A_{\text{ACR}}) \setminus \{e\}$ was open in G^A_{ACR}, then it is closed in T_j. Therefore, we have to close the single open (optional) edge in the original path in G corresponding to d. Thus, if we close an edge with respect to G^A_{ACR}, then it is certain which original edge in G^A we have to turn on.

By contrast, if $d \in (T_j \triangle G^A_{\text{ACR}}) \setminus \{e\}$ was closed in G^A_{ACR}, then all edges in the original path corresponding to d were closed in G^A. As $d \notin T_j$, we have to open precisely one edge of this original path in the reconfiguration. Concluding, we have again a choice of which edge we open.

Let still $e \in G^A_{\text{ACR}} \setminus T_j$. For the sake of completeness, for each original edge corresponding to e, we try each combination of possible edges to open given by the path edges in $(T_j \triangle G^A_{\text{ACR}}) \setminus \{e\}$, until we find a possible reconfiguration accepted by the load flow. We explain this in more detail by means of the example. We display G^A_{ACR} and some new spanning tree T_j, respectively:





Here $G^A_{\text{ACR}} \setminus T_j = \{5\}$, so take $e = 5$. Then $(T_j \triangle G^A_{\text{ACR}}) \setminus \{e\} = \{2\}$. Using the original graph and the subdivision into categories, we see that the original edges corresponding to the path edge 5 are the edges 5 and 6. Similarly, the original edges corresponding to the path edge 2 are the edges 2 and 4. Using the above, we know we can find at least one switchover using $(T_j \triangle G^A_{\text{ACR}}) \setminus \{e\}$ for both edge 5 and edge 6, as we can choose which one to open. A switchover consists of turning on or off original edges corresponding to $(T_j \triangle G^A_{\text{ACR}}) \setminus \{e\}$. As path edge 2 was open in G^A_{ACR}, we have to close the optional edge in the corresponding original path. This is the optional edge 4. Thus, we can switch original edge 5 using original edge 4 and switch original edge 6 using edge 4 too.

Note that we do not have a choice between switchovers in this example. Here we only have a choice in the edges that we switch. In other cases, we could have more choices, which we would then check one by one, until we have found a switchover for all original edges corresponding to the path edges in $G^A_{\text{ACR}} \setminus T_j$, or until we have tried all choices.

Combining switchovers from the Andrei-Chicco reduction and spanning trees
We mention a point of attention with regard to all possible switchover combinations. This section explained both how to find switchovers using the Andrei-Chicco reduction itself and how to find switchovers using new spanning trees (in the Andrei-Chicco reduction). However, it could be useful to combine switchovers from both methods. Especially if the physical conditions did not yet accept a switchover for a particular edge, another possibility could lie in the combination of a switch within a path (or loop) and a switchover given by a new spanning tree. Nonetheless, we must never switch more than k edges.

First note that we combine these switchovers as combinations of edges in the original graph. Thus, if we computed the switchover information using the Andrei-Chicco reduction and using new spanning trees, we combine the findings afterwards.

Second, without using the Andrei-Chicco reduction, we did not have to worry about forgetting certain possible switchovers. For all possible switchovers were captured in the spanning tree strategy. However, by the Andrei-Chicco reduction we could both switch within a path (or loop) and between different path edges using spanning trees. As we consider these findings only separately, it is important to combine them afterwards. Only then we are sure we did not skip possible switchovers.

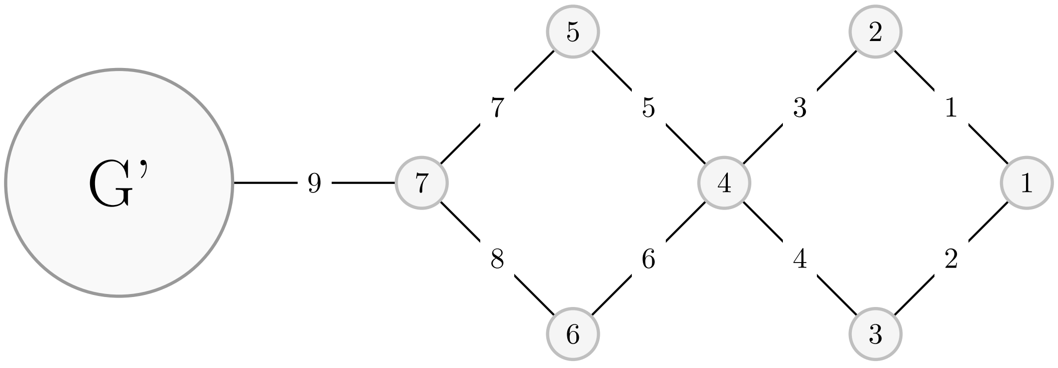
Further note that we could aggregate switches within loops, paths and across different paths (using spanning trees) in any combination, provided the combination preserves that the reconfiguration of the network is a spanning tree and does not switch more than k edges. For example, we could combine two switchovers coming from loops, or two switchovers from paths, or one from a loop with one from a spanning tree, etcetera. In the particular case where we may switch only one edge ($k = 1$), we clearly cannot combine these switchovers and so we do not have to.

We further discuss this important notion of combining switchovers in chapter 6, in particular in subsections 6.2.4 and 6.3.1, where we really add this notion in the eventual algorithm. There it is important to limit the computation time of these additional possibilities as much as possible. It would be a pitfall to repeatedly compute the same combinations, or to forget some of the combinations. As the required computation of combinations depends on the order of the reductions and algorithms used and the number k, we only describe this in chapter 6, where we deal with the implementation of the check of the $m - 1$ principle in R.

If we again look at the example on page 36, it could happen that switching edge 7 using edge 9 (switching within a loop) is not accepted by the physical conditions, even as switching edge 5 using edge 4 (switching using a new spanning tree). However, perhaps it is acceptable to switch edge 7 by turning on or off edges 9, 5 and 4. Then we have a switchover that is a combination of a switch within a loop and a switch coming from a new spanning tree. Note that there are multiple combinations like this that are possibly suitable.

Considerations on repeating the Andrei-Chicco reduction

We now come back to the possibility of reducing the reduced graph. After the completion of the reduction using the principles sequentially, we could perhaps use some of the three principles on the reduced graph again. Looking at example 3.17, we see that after the reduction we could again apply principle 2, for the removal of the loop has created a vertex of degree two, namely vertex 1. Vertex 1 forms a reducible path with starting point 2 and end point 4. In general, it could take any number of applications of the principles before no reduction is possible any more. If we look at a graph of the following shape for example:



Here G' is some complex graph, meaning that G' does not disappear completely after using the reduction principles several times. If we apply principle 2, node 1, 2, 3, 5 and 6 disappear, leading to two edges between the remaining vertices 4 and 7 and a loop situated at vertex 4. After applying principle 3, the loop disappears and 4 is of degree two. Hence we can apply principle 2 again, giving a loop situated at vertex 7. After using principle 3 again, vertex 7 is just a leaf. After using principle 1, only the part G' remains. One can check for oneself that performing these steps in this order gives this result.

This graph is just an example, but leads to the understanding that we could use the principles multiple times such that the remaining graph reduces significally. We can even go further and state that G' could also have the given shape of the example graph, so we could apply the principles once again multiple times. As G' has to be finite but could be arbitrarily large and therefore could have this shape an arbitrary number of times, we discover that we could use the three principles an arbitrary number of times on certain graphs.

3.4 An algorithm to find all spanning trees

An often cited article giving an algorithm to find all spanning trees of a given connected undirected graph is the article by Gabow and Myers (1978) [7]. Their algorithm finds spanning trees in depth-first search order (see section 2.2), has time complexity $O(n + m + n \cdot s)$, where s equals the number of spanning trees in the graph, and has space complexity $O(n+m)$. As may be clear, we are interested in the time complexity, as this is the mathematical way to express the speed of an algorithm. Recall the definition of time complexity in section 2.4.

Gabow and Myers (1978) state that their algorithm is optimal with respect to the time complexity. Later articles (Matsui (1993) [8], Kapoor and Ramesh (1995) [9], Shioura and Tamura (1995) [10]) confirm this, however, noting that the time complexity can be lowered if one not explicitly needs a list of all spanning trees. The optimal time complexity is $O(n+m+s)$ for an algorithm enumerating all spanning trees in compact form, i.e. an algorithm listing only the relative changes (different edges) between two consecutive spanning trees found, instead of listing all entire spanning trees (given by their edges). We would like to know the entire spanning trees, as this simplifies the comparison with the initial configuration a lot. Therefore, we will use an algorithm that outputs all spanning trees and has time complexity $O(n + m + n \cdot s)$.

Several other algorithms that explicitly output all spanning trees also have time complexity $O(n + m + n \cdot s)$. The algorithm by Matsui (1993) is more flexible than the algorithm by Gabow and Myers (1978), in the sense that one can adapt the algorithm such that it finds the spanning trees in depth-first order, breadth-first order or in best-first order if the graph is weighted. In chapter 6, *Implementation & Overview*, particularly subsection 6.2.3, we will see that the number of spanning trees of the k reduced and Andrei-Chicco reduced graph is small, wherefore this limitation to the algorithm by Gabow and Myers does not matter that much for the overall algorithm (to check the $m - 1$ principle). Conversely, the algorithm by Gabow and Myers is less complex than the algorithm by Matsui and as a result easier to implement in a programming language.

In this section, we describe the algorithm and mathematical background given by Gabow and Myers (1978). First, we give some definitions and present the algorithm. Second, we prove some graph-theoretically properties assumed in the algorithm. Last, we prove the time complexity of the algorithm is indeed $O(n + m + n \cdot s)$.

3.4.1 The algorithm by Gabow and Myers

As mentioned before, for a given connected graph G, the algorithm by Gabow and Myers (1978) [7] prints a list that contains each spanning tree exactly once. Gabow and Myers (1978) present their algorithm for directed graphs, but give an adaption to use it for undirected graphs too. We present here the algorithm for directed graphs and subsequently explain how to use it on undirected graphs.

Let a connected directed graph G with root r be given. A spanning tree (rooted at r) of G is a subgraph having a unique directed path from r to any vertex of G. An edge e is a bridge if $G \setminus \{e\}$ is not rooted at r (so $G \setminus \{e\}$ can still be connected in some sense, but not such that there exists a path from r to each other vertex). Then the bridge e is in every spanning tree.

The goal is yet to find all spanning trees rooted at r. We describe the general approach by Gabow and Myers (1978). At the end of this subsection, we display an example (example 3.18).

The approach

Suppose we have a given tree T rooted at r. We want to find all spanning trees containing T. Let $\partial(T)$ be the directed edge cut of G associated with T, i.e. the set of edges $\{e_1, e_2, \ldots, e_l\}$ directed from a vertex in T to a vertex not in T. We can first try to find all spanning trees containing $T \cup \{e_1\}$ and then delete e_1 from the graph (as we already have all spanning trees containing $T \cup \{e_1\}$). Afterwards, we can search for all spanning trees containing $T \cup \{e_2\}$ (in the modified graph, so without e_1) and then delete e_2. To continue, we repeatedly choose an edge e_i from $\partial(T)$. We search for all spanning trees containing $T \cup \{e_i\}$ (in the modified graph) and then delete e_i. We could repeat this until we have done this for all edges in $\partial(T)$, but we take a somewhat smarter approach to this.

Assume that we have repeated this procedure up to edge $e_k \in \partial(T)$. Suppose e_k is a bridge in the modified graph. Then e_k is in every spanning tree of the modified graph, so all spanning trees containing T of the modified graph are already in the set spanning trees containing $T \cup \{e_k\}$ of the modified graph. To conclude, the first time we find an $e_k \in \partial(T)$ that is a bridge in the modified graph, we have found all spanning trees containing T. Moreover, we have found all spanning trees containing T exactly once. As a spanning tree found in the step using e_i does not contain any of the edges $e_1, \ldots, e_{i-1}$, but the spanning trees found in the previous steps did contain at least one $e_j \in \{e_1, \ldots, e_{i-1}\}$.

If $T = \{r\}$, we would find all spanning trees rooted at r. However, to implement the idea outlined in practice, we should recursively use this idea on subgraphs and subtrees. We will explain this in more detail while presenting the algorithm in pseudo code. For now it is enough to know that T will grow step-by-step recursively.

Depth-first search

Given the approach, we need an efficient method to detect whether an edge e_k is a bridge. Suppose that we find all spanning trees in depth-first order. We will clarify how this simplifies the detection of a bridge.

We let T grow as depth-first as possible, i.e. we add the edge $e = (u, v)$ to T that gives the greatest depth (if we have a choice). Formally, this means the vertex v added to the tree T is one which is a neighbour of as recent an addition to T as possible, as described in section 2.2.

Suppose we have found all spanning trees containing $T \cup \{e\}$ and we need to test whether $e = (u, v)$ is a bridge. If L is the last spanning tree found that contains $T \cup \{e\}$, then v has the fewest descendants possible among all spanning trees containing $T \cup \{e\}$. This is intuitively clear, as the last spanning tree found should be the least depth-first. In lemma 3.21, we will prove that e is a bridge *iff* no edge goes from a non-descendant of v to a proper descendant of v. (Here the descendant relationship is based on L, but we test the property on the modified graph.) Then we can simply check this property to see whether e is a bridge. This gives an efficient bridge test, according to Gabow and Myers (1978). Thus, if we find all spanning trees in depth-first order, we have a suitable bridge test. We now explain the depth-first search in greater detail.

In the algorithm, we will use F to represent $\partial(T)$, i.e. as a list of all edges directed from vertices in T to vertices not in T. F should treat vertices on a last-come first-served basis, as we need a depth-first tree search. Thus F needs to behave as a stack: new edges are pushed onto the top of F and an edge e to enlarge T is popped from the top too. Besides this, some other edges are removed from F when e is removed from F and added to T, because $\partial(T)$ changes if T changes. However, at a later stage, e is removed from T (when we have found all spanning trees containing $T \cup \{e\}$) and F should be identical to F *before the pop of* e (except for e as e is removed from G). In other words, at that point, all edges that we removed from F when e was added to T need to be at their original place in F as before. To conclude, the removal and restorage of edges in F should leave the order of edges in F unchanged. This warrants the tree-search is depth-first.

The algorithm

Below, we display the pseudo algorithm as presented by Gabow and Myers (1978) [7]. As said, T is the current tree of G that we extend in recursive invocations of the algorithm until we find a spanning tree. Thereafter T shrinks. T will expand to another spanning tree later on, then it will shrink again, and so on. If we execute a recursive call of GROW, we take the most recent values of G, r, T and F as its input arguments.

In the paper, T is represented as a real subgraph, so as a combination of vertices and edges. In the computer program in R, we represent T as a set of edges. This determines the vertices too as T is connected. (Except in the initial case where T consists of only one vertex and hereby no edge. Though, by defining $F = \partial(T)$ as the directed edges starting in this vertex, T is determined too.)

As mentioned before, F represents $\partial(T)$ and leaves the order of edges unchanged by removal and subsequent restorage of the same edges. Besides F, local lists FF are used to reconstruct F after a recursive invocation. Lastly, we denote by L the last spanning tree found thus far. This spanning tree L is given by its edges (like T).

```
GROW: function to find all spanning trees rooted at $r$ containing $T$
Input arguments: $G$ (given by its edges), $r$, $T$, $F$

 1.                      if $T$ has $n-1$ edges then $L := T$; output($L$);
 2.                      else make $FF$ an empty list, local to GROW;
 3.                          repeat
 4. new tree edge:               pop an edge $e$ from $F$, let $e$ go from $T$ to vertex $v$,
                                 $v \notin T$;
 5.                              add $e$ to $T$;
 6. update F:                    push each edge $(v, w)$, $w \notin T$, onto $F$;
 7.                              remove each edge $(w, v)$, $w \in T$, from $F$;
 8. recurse:                     GROW;
 9. restore F:                   pop each edge $(v, w)$, $w \notin T$, from $F$;
10.                              restore each edge $(w, v)$, $w \in T$, in $F$;
11. delete e:                    remove $e$ from $T$; remove $e$ from $G$; add $e$ to $FF$;
12. bridge test:                 if there is an edge $(w, v)$, where $w$ is not a descendant
                                 of $v$ in $L$ then $b :=$ FALSE; else $b :=$ TRUE;
13.                          until $b$
14. reconstruct G:           pop each edge $e$ from $FF$, push $e$ onto $F$, add $e$ to $G$;
```

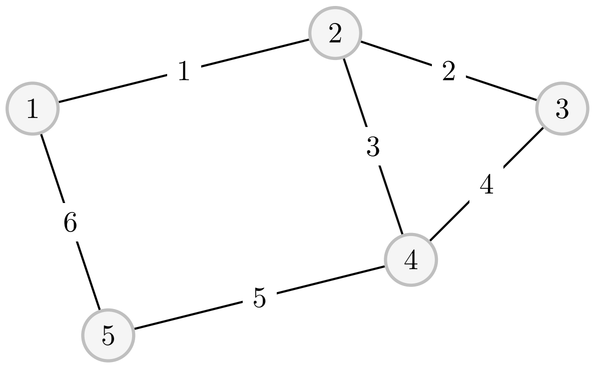
To find *all* spanning trees rooted at r, we need the following two initial declarations: let T only contain the root r and let F contain precisely all edges (r, v). Then invoking GROW results in a list of all spanning trees rooted at r. Note that in the program R we represent T by its edges only. There we define T initially empty, but F should still contain all edges (r, v). In the first invocation of GROW, the first edge in F is added to T. From then on, the representation of T in R is similar to that in the pseudo code.

We note another little difference in the pseudo code compared to the implementation in R. In R, the edge e is removed from G before line 10 instead of in line 11. As a result, we do not restore edge e in F in line 10. We should not do that, as we already found all spanning trees containing $T \cup \{e\}$ in the recursive call(s) in line 8. Only when T is shrunk, we eventually want to use e again. This is provided by the use of FF that will restore e after the repeat loop.

The undirected case

We can adapt the method to find all directed spanning trees rooted at r easily to the undirected case of finding all spanning trees. We make an undirected graph G directed by giving each undirected edge both directions, so splitting each edge in two directed edges in opposite directions. It may be clear that both representations (directed and undirected) are equivalent. Furthermore, the root r can be arbitrarily chosen, as an undirected tree could have any node as root, but the root is not relevant at the same time. With these two modifications, the function GROW finds all spanning trees of G. We prove the correctness of the algorithm in the next subsection.

Example 3.18. An undirected graph G ($r = 1$) and the result of the invocation of GROW. We display all eleven spanning trees (given by its edge numbers) in depth-first order:



1 2 4 5
1 2 4 6
1 2 3 5
1 2 3 6
1 2 5 6
1 3 4 5
1 3 4 6
1 4 5 6
2 3 5 6
3 4 5 6
2 4 5 6

In this example, the first invocation of GROW calls 25 other (recursive) invocations of GROW in total.

3.4.2 Proofs: Correctness of the algorithm

In this subsection, we proof that the function GROW behaves correctly in the sense that it finds all spanning trees of a given directed or undirected graph. To do this, we first prove that the tree T grows in depth-first order. Afterwards, we show that the implementation of the bridge test (using the descendant relationship) is correct. Finally, we prove that GROW functions properly.

Before showing that the tree T grows in depth-first order, we first prove in the following lemma that F manages the edges on the border of the tree T and keeps them in depth-first order.

Note that the original graph G is modified in the execution of GROW, as in lines 11 and 14 of the pseudo algorithm edges are removed and replaced, respectively. For this reason, we call the state of G at a certain point in the computation the *present* graph G.

Lemma 3.19. *Let* GROW *be invoked with input arguments G, r, T and F. In particular, F contains the sequence of edges (v_i, w_i), $i = 1, \ldots, |F|$. (The edge (v_1, w_1) is on the top of the stack F.) Then we have:*

(i) $F = \partial(T)$, *i.e.:*
$\{(v_i, w_i) \mid 1 \leq i \leq |F|\} = \{(v, w) \mid v \in T,\ w \notin T,\ (v, w)$ *is in the present graph* $G\}$.

(ii) *F contains the edges in depth-first order, i.e.:*
If $j \leq i$, then v_j is a descendant of v_i in T.

Proof. In the first place, note that F on exit from GROW is identical to what F was on entry. This is because the changes to F in the beginning of GROW are undone in later parts of GROW. Specifically, lines 4, 6 and 7 are undone by lines 14, 9 and 10, respectively. Furthermore, note that T on exit from GROW is identical to what it was on entry, as the only changes to T are made in lines 5 and 11 and they cancel each other.

Next we proof clauses (i) and (ii) at the same time, using induction. The base case: In the first call to GROW, we have $T = \{r\}$ and F contains precisely all edges of the form (r, v). Then $v \notin T$ as the graph G cannot contain loops, so (i) holds. (ii) holds too, as each $v_i = r$ and a vertex is a descendant of itself.

The inductive case: Suppose (i) and (ii) hold on entry of a recursive call of GROW. By the preliminary remark, we know they hold on exit from this computation of GROW too, because T and F are on exit identical to their input values. We only need to check that (i) and (ii) hold for the input values for each new recursive call to GROW in line 8. We first assume this new recursive call is in the first iteration of the repeat loop. We come back to the other cases at the end of this proof. In the first iteration of *repeat*, the differences of T and F in line 8 compared to their input values are as follows: T is extended with the first edge $e = (v_1, w_1)$ of F (line 5), e is removed from F (line 4), all edges (w_1, w), $w \notin T$, are added to F (line 6) and all edges (w, w_1), $w \in T$, are deleted from F (line 7). To prove that (i) holds: we need that $F = \partial(T)$ for the new T and F. The differences between the old and new $\partial(T)$ are: (1) edges ending in w_1 (so e too) need to be removed, as w_1 is now part of T, and (2) edges starting in w_1 that do not end in the new T should be added to F, for the same reason. This is exactly what is done in lines 4, 6 and 7. Thus, (i) holds.

We need to check property (ii) only for the new (added) edges, as by the induction hypothesis (ii) holds for the other edges. Of course, it does not matter for (ii) that some of the 'old' edges are removed now. Each new edge is of the shape (w_1, w), $w \notin T$. All these edges are added at the top of F. As these edges all have starting point w_1, (ii) holds pairwise for the new edges. Take a new edge (w_1, w) and an edge (v_i, w_i) that was already in F on entry of GROW. w_1 is a descendant of v_1 in T, as e is in T. Besides, v_1 is a descendant of v_i as the 'old' F has property (ii). Combining these observations and knowing that the relation 'descendant of' is transitive, we have that w_1 is a descendant of v_i in T. To conclude, (ii) holds for all edges in the new F.

Lastly, we have hitherto ignored the presence of the repeat-until loop. This means we have proved the lemma's correctness if line 3 and 13 did not exist. We now know (i) and (ii) hold for the first recursive call of GROW (in line 8) and we will show they also hold for recursive calls of GROW in later iterations of the repeat loop. The only difference between the first and second iteration of the repeat loop is the absence of the edge e that was popped from F in the first iteration. Except this edge, everything is restored at the end of the first iteration as before the first iteration. The edge e does not exist in the present graph of the second iteration, but that does not really matter: (ii) still holds with the removal of an edge of F. Furthermore, e does not exist in all G, F and T, so (i) still holds too. Therefore, by induction, (i) and (ii) hold in each iteration of the repeat loop and with the previous observations for all calls of GROW. □

A useful corollary of the lemma asserts that T is a depth-first tree at any point in the computation of GROW:

Corollary 3.20. *Let e_j, $1 \leq j \leq |T|$, be the edges in T, indexed in the order they are added to T. Let $e_j = (w_j, v_j)$. Then the descendants of any v_j in T are vertices v_k, $j \leq k \leq J$, for some J.*

Proof. It suffices to proof that the descendants of a given vertex v are added to T consecutively (starting with v itself, as v is a descendant of v). If v has no descendants but itself in T, this is of course true (then $J = j$). Suppose therefore that v has at least two descendants in T. When v is added to T (via an edge e), edges (v, w) are pushed onto the top of F (line 6). There is at least one such edge pushed, for otherwise v has no descendants but itself in the end: If there is no (v, w) to add to F at that moment, there is never a (v, w) to add to F. Since such edges can only be dropped from F in later recursive calls (in line 7) and not added as $v \in T$ yet.

Knowing there are edges (v, w) pushed onto the top of F, lemma 3.19(ii) now shows that as long as any of those edges (v, w) is in F, the edges added to T join descendants of v (as well as when new edges are pushed onto the top of F in other recursive calls). When the last edge (v, w) is removed from F, lemma 3.19(i) shows that there are no edges from descendants of v to vertices not in T, as they would be pushed onto F otherwise and (v, w) would not be removed. (Use the same reasoning as before: when T grows, there will only be fewer edges of that shape.) Thus, no other edges can have an end that becomes a descendant of v than those that were consecutively added after v. □

For the sake of clarity, we now know that T grows in depth-first order. We continue by proving that the bridge test (implemented in line 12) is correct. For this we can assume now that we find the spanning trees in depth-first order.

Lemma 3.21. *The bridge test (in line 12) sets b to* TRUE iff *edge $e = (u, v)$ is a bridge of the present graph G. In other words, there is no edge $(w, v) \neq e$ in G, where w is not a descendant of v in L,* iff *e is a bridge of the present graph G.*

Proof. Let $e = (u, v)$ and let D_v denote the descendants of vertex v in the latest spanning tree L. Clearly, $r \notin D_v$. We will use the following claim:

Claim: The present graph G has no edge (w, x), where $w \notin D_v$, $x \in D_v \setminus \{v\}$.

We first explain how this claim proves the lemma and we proof the claim afterwards. e is not a bridge iff there exists some path P not containing e from r to v (see section 2.2). Suppose e is not a bridge and we have such a path P. If there are no edges (w, x) as above, P must end in an edge (w, v), $w \notin D_v$. Conversely, if e is a bridge and there is no path without e from r to v, then $G \setminus \{e\}$ consists of the components D_v and $G \setminus D_v$, if we accept the claim. If there were an edge (w, v), $w \notin D_v$, then this edge would connect the separate components, which is not the case. Thus, the claim proves the lemma.

Currently, it suffices to prove the claim. Let L, the last found spanning tree, have edges e_j, $j = 1, \ldots, n-1$, in the order we added them in the formation of L. Let $e = e_i$. By the recursion, the bridge test was already executed on edges e_j, $j = n-1, \ldots, i+1$, and now we test $e_i = e$. Then we know: for $j > i$, this bridge tests set b to TRUE, for otherwise, another spanning tree would be found after L (we would perform another iteration of the repeat loop and find another spanning tree).

Consider any vertex $x \in D_v \setminus \{v\}$. By corollary 3.20, the edge in L directed to x must be some e_k, $k > i$. Then the bridge test for e_k is set to TRUE. Therefore, no edge (w, x), $w \notin D_v$, exists when that bridge test is executed.

Then an edge (w, x), $w \notin D_v$, is in the present graph only if it is added in an execution of line 14 (by "add e to G") following the bridge test of some e_l, where $i < l \leq k$. By corollary 3.20, e_l has descendants of v as its ends. The edges added after the bridge test of e_l precede e_l in the list F (by the way we use FF in lines 11 and 14). By lemma 3.19(ii), these edges start in D_v. In summary, no edge (w, x), $w \notin D_v$, is added. Thus no edge (w, x), where $w \notin D_v$, $x \in D_v \setminus \{v\}$, is in the present graph. □

As we have yet proved the correctness of the bridge test, we can now prove that the function GROW finds all spanning trees rooted at r, if it has the prescribed initial declarations.

Theorem 3.22. *The function* GROW*, having as input: a directed graph G rooted at r, T only containing the root r and F containing exactly all edges (r, v), finds all spanning trees rooted at r of G.*

Proof. Let GROW be called with T a tree rooted at r and let A be the present graph when GROW is called. We claim that it suffices to show that GROW finds all spanning trees rooted at r of A containing T. For in the initial call of GROW, T contains only the root r and $A = G$.

We proof the claim by induction, where we order the calls to GROW to assure that the size of T is non-increasing. The base case: T contains $n-1$ edges. Then line 1 behaves correctly, as there is exactly one spanning tree containing a (spanning) tree having $n-1$ edges.

The inductive case: suppose T contains less than $n-1$ edges. Let F contain edges e_i, $i = 1, \ldots, |F|$. We define:

$\mathcal{T}_i = \{R \mid R$ is a spanning tree rooted at r and $T \cup e_i \subseteq R \subseteq A \setminus \{e_j \mid 1 \leq j < i\}\}$.
By induction we can show that GROW finds the trees in $\bigcup_{i=1}^{k} \mathcal{T}_i$, where e_k is the first edge for which the bridge test sets b to TRUE (in line 12). For we leave GROW shortly after this first (and only) positive bridge test. In the i-th iteration of the repeat loop, for $1 \leq i \leq k$, GROW finds $\mathcal{T}_i$, as in that iteration e_i is popped from F and added to T.

To fill in the last details, lemma 3.21 proofs e_k is a bridge in the graph $A \setminus \{e_j \mid 1 \leq j < k-1\}$. Thus, any spanning tree R of A that contains T contains some e_j, $1 \leq j \leq k$, so $R \in \mathcal{T}_j \subseteq \bigcup_{i=1}^{k} \mathcal{T}_i$. To conclude, GROW finds the desired spanning trees. As sets $\mathcal{T}_i$ are disjoint, GROW finds each spanning tree exactly once. □

Knowing the correctness of Gabow and Myers' algorithm, we can proceed by estimating the algorithm's efficiency. We do this in the next subsection, using the time complexity.

3.4.3 Proofs: Complexity of the algorithm

We need to give some implementation details before we can estimate the complexity of GROW. First, we discuss how F is managed, and second, we discuss how the bridge test is performed.

F should be a kind of doubly linked list of edges. Line 7 of the pseudo code traverses the list of edges directed to v, from beginning to end. Each edge directed from T to v is deleted from F. The values of the links are not removed nonetheless. Line 10 traverses the list of edges directed to v in reverse order. Here, each edge directed from T to v is inserted back in F, at the position given by its link values. In this way, each edge is restored in its original position. In the implementation in the program R, we do not really remove and insert the edges in F, but let F behave as a stack where each item consists of two elements: the edge number and a boolean that represents whether the edge is in use (really in the stack) or not (temporarily removed).

Now we discuss the implementation of the bridge test. We can efficiently detect descendants by numbering the vertices in preorder associated with L. To explain this, we give some definitions and a lemma that relates a preorder with the descendant relation. These definitions and observation are derived from the book *The Design and Analysis of Computer Algorithms* by Aho, Hopcroft and Ullman (1974) [11], pages 53-55.

Definition 3.23. Let T be a tree having root r and let $v_1, \ldots, v_k$, $k \geq 0$, be all vertices such that $(r, v_i) \in T$. Note that a vertex v with all its descendants is called a *subtree* of T, whose root clearly is v. A **preorder traversal** P of T is defined recursively as follows:

1. Visit the root r first.

2. Visit in preorder the subtrees with roots $v_1, \ldots, v_k$ in that order.

We simply call P a preorder. We regard P as a function on the vertices of T: for a vertex v that is visited as i-th vertex, $P(v) = i$. Then we define the function H on the vertices in the following way: $H(v)$ is the highest preorder number of a descendant of v.

Remark 3.24. As one can see: $P(r) = 1$ and $P(v_1) = 2$. If w_1 is the root of the first subtree of the subtree with root v_1, then $P(w_1) = 3$, etc.
Another important observation is that all vertices in a subtree with root v have preorder number no less than v.

We have the following property that combines the preorder numbers with the descendant relationship. It follows clearly from the definition and remark.

Lemma 3.25. *Let T be a tree and v a vertex in T. If D_v is the set of descendants of v, we have: $w \in D_v$ iff $P(v) \leq P(w) < P(v) + |D_v|$. In other words, if vertex v has $|D_v|$ descendants, its descendants have preorder numbers $P(v)$, $\ldots$, $P(v) + |D_v| - 1$.*

Proof. By induction on the definition of a preorder traversal of T. □

It may be clear that the lemma provides an easy check to determine whether a vertex is a descendant of another vertex in a tree, if we know a preorder of the tree. Specifically:

$$w \text{ is a descendant of } v \textit{ iff } P(v) \leq P(w) \leq H(v). \tag{3.2}$$

Notice that $H(v) = P(v) + |D_v| - 1$. (3.2) gives us an efficient test as part of the bridge test in line 12. We compute and store the values of P and H in line 1, when L is formed.

Note that after the initial assignment of a preorder P and H, we can answer the question of whether w is a descendant of v in a fixed amount of time, independent of tree size.

Although we are not interested in the space complexity, we need to know the space complexity to compute the time complexity. Fortunately, the computation of the space complexity is quite straightforward.

Theorem 3.26. *Let G be a directed or undirected graph having n vertices and m edges. All spanning trees of G can be found by* GROW *in space $O(n + m)$.*

Proof. Note that we do not save the spanning trees found, but only output them. Therefore, we do not include the space needed for the spanning trees in the space complexity. Ideally, the graph G is stored as a collection of doubly linked lists of edges to and from each vertex. Then this uses $O(m)$ space. In the computation, an edge e may be on the list F or on at most one list FF (if e is in some FF, then it is not in F and hereby cannot be put on another FF list). For this reason, F and all lists FF together use only $O(m)$ space. For T, P and H, we need $O(n)$ space, as should be evident. Combining all of this, the space complexity is $O(n + m)$. □

In connected graphs: $n \leq m + 1$, so in practice, the space complexity of Gabow and Myers' algorithm approximates $O(m)$.

Knowing the space complexity, the implementation of F and the implementation of the bridge test (given by the descendant relationship, in turn given by the preorder), we can prove the time complexity of the algorithm on undirected graphs:

Theorem 3.27. *Let G be an undirected graph having n vertices, m edges and s spanning trees. All spanning trees of G can be found by* GROW *in time $O(n + m + n \cdot s)$.*

Proof. We estimate the time spent on each line of the pseudo algorithm. To begin with, line 1 does a preorder traversal and outputs each spanning tree. As the preorder traversal visits each vertex once and the spanning tree found has $n - 1$ edges, the time needed in one execution of line 1 is $O(n)$. We execute line 1 once for each spanning tree found, so we spend $O(n \cdot s)$ time in line 1 in total (by multiple calls of the recursive function).

Now we analyze the time spent in lines 4-12 for one particular edge $e = (u, v)$ added in line 4. However, we temporarily ignore the recursive call in line 8. The time spent in these lines is proportional to the number of edges incident to v, as we need to store and delete these edges. We note that the time spent in lines 4, 5 and 11 is $O(1)$. As we already computed P and H in line 1, we only need to compare the values $P(v)$, $P(w)$ and $H(v)$ in line 12, which is $O(1)$ for a single vertex w. As there are at most $n - 1$ edges ending in v, $O(n)$ time is spent in line 12 on all edges (w, v) in total. Thus, we spend at least $O(n)$ time in lines 4-12 for one edge $e = (u, v)$ added in line 4. Now we only need to focus on the time devoted in lines 6, 7, 9 and 10. We distinguish two cases and show that both cases lead to a total time of $O(n \cdot s)$ spent in lines 4-12:

1. Suppose e turns out to be a bridge after the bridge test (in a particular present graph A).
   Then each edge f that has v as one of its ends is in some spanning tree R containing $T \cup \{e\}$. We may assume the time spent on f in R is $O(1)$. Hereby the time spent on R is $O(n)$. Concluding, the total time spent on bridges in lines 4-12 is $O(n \cdot s)$.

2. Suppose e is decided not to be a bridge after the bridge test (in a present graph A).
   We look at the time spent in lines 4-12 (actually, only at lines 6, 7, 9 and 10) in a certain iteration in which e is popped from F. We need to push, pop, remove and restore particular edges with end v. As there are at most n edges connecting v with some other vertex, the time spent on these edges is $O(n)$. Thus, the time spent in lines 4-12 in a given iteration in which e is popped is $O(n)$.
   We will now proof that there are exactly $s - 1$ 'nonbridges', edges that turn out to be no bridge in the present graph immediately after the bridge test. Let a nonbridge e correspond to the tree L used in the bridge test of e. As e is not a bridge, e will be deleted and another spanning tree is grown before the next bridge test of another edge (in the algorithm we have another iteration of the repeat loop). Therefore, a given tree L corresponds to at most one nonbridge, i.e. at most one bridge test with outcome 'no bridge' is done on some tree L, as this outcome immediately leads to the production of a new L. On the other hand, if

L is a spanning tree but the last one found (after the termination of GROW), it is used in the bridge test for some nonbridge, for otherwise no new spanning tree will be found (but this is the case). Thus, each spanning tree but the last one corresponds to precisely one nonbridge.
We conclude that the total time spent in lines 4-12 on nonbridges is $O(n \cdot s)$, as we have $s - 1$ nonbridges and each nonbridge uses $O(n)$ time in these lines.

Thus, for both bridges and nonbridges (after the bridge test in line 12) we need $O(n \cdot s)$ time in lines 4-12. The total time spent in lines 4-12 is then $O(n \cdot s)$ too. As we already took into account all executions of lines 4-12 (although caused by the repeat loop and recursion), we can neglect the time spent in lines 2, 3, 13 and 14, as these lines itself take only $O(1)$ time.

Lastly, we may assume the allocation of a given amount of memory takes a proportional amount of time too. Hence we need at least $O(n+m)$ time to allocate the memory in the algorithm, by theorem 3.26. Combining this with the 'actual' computation time required in the algorithm, we conclude that the time complexity is $O(n+m+n \cdot s)$. □

In general, we have a lot of spanning trees in a connected undirected graph. For section 3.2 showed that the number of spanning trees grows very rapidly with the number of edges for a fixed number of nodes. As a consequence, the time complexity of the algorithm is very dependent on the number of spanning trees s.

A last note concerns why the time complexity of the algorithm is optimal for the exercise of finding all spanning trees. We noticed that $O(n \cdot s)$ time is required for the output of all spanning trees as lists of edges. The factor $O(n + m)$ can be neglected for almost all cases, because s is many times greater than n and m as just seen. Thus, the algorithm is optimal to within a constant factor.

Chapter 4

Physical Background

To model a real electricity distribution grid, one should take into account the rules of physics regarding electricity. The first section mentions some basic rules concerning quantities as current, voltage, resistance and power. The way to perceive these quantities as complex measures is explained too. The second section presents a commonly used method to model and calculate the values of these quantities in an electricity distribution grid. This method is called the *load flow model.* The third section gives an alternative to the load flow model, called the *linear model.* This model approximates the values of the physical quantities in the electricity distribution network. The linear model is recently developed at Alliander and computes the values much faster than the load flow model. Lastly, we describe the application of the load flow model or linear model in the $m-1$ problem, especially how we use one of the models to check the current and voltage capacities: whether a current or voltage limit is exceeded in a new configuration.

4.1 Electricity physics

In an electricity distribution grid, power P is transported from bus (node) to bus, through a cable with a certain resistance R. This power is caused by a voltage difference ΔU between the buses, inducing a current I through the cable between the buses. To give an overview of the important quantities and associated units in the physics of electricity distribution networks:

Quantity	**Abbreviation**	**Unit**	**Abbreviation**
Voltage	U	Volt	V
Current	I	Ampère	A
Resistance	R	Ohm	Ω
Power	P	Watt	W

Two basic physical laws concerning electricity are:

Ohm's law: $R = \dfrac{\Delta U}{I}$ **Joule's law:** $P = \Delta U \cdot I$

In general, one perceives these quantities as complex numbers. The voltage, current and power have a real and a reactive (imaginary) component. The word resistance, however, is used for the real physical quantity. The complex ratio $\frac{U}{I}$ is called the *impedance* Z, where the real component is the *resistance* and the complex component the *reactance*. In this way, one can consider the values of physical quantities as vectors in the complex plane. To understand this vision, one first needs to understand the generation of alternating current, which is explained in the lecture notes *Wisselstroom* by Christianen (2014) [22]. A more extensive description of alternating current and complex physical quantities can be found in the book *Physics for Scientists and Engineers* by Serway and Jewett (2008) [17].

4.1.1 Alternating current

One generates alternating current (AC) by spinning a coil in a magnetic field, illustrated in figure 4.1. In the coil, the spinning raises a time dependent voltage $U(t)$, where:

$$U(t) = U_0 \cos(\omega t + \phi_U) \ .$$

U_0 is called the *voltage amplitude* and ω is called the *angular velocity*, the speed of the spinning coil. ϕ_U comprises the *voltage angle*, the starting angle relative to the magnetic field. The voltage $U(t)$ depends on the alternating angle between the coil and the magnetic field. We assume the angular velocity is perpendicular to the magnetic field.

Suppose the coil is connected to a power circuit with resistance R. Then the time dependent current $I(t)$ equals (using Ohm's law):

$$I(t) = \frac{U_0}{R} \cos(\omega t + \phi_U) = I_0 \cos(\omega t + \phi_I) \ .$$

Here $I_0 = \frac{U_0}{R}$ and $\phi_I = \phi_U$. I_0 is called the *current amplitude* and ϕ_I indicates the *current angle*. In general, the voltage angle and current angle can be different, this will be clear in the complex notation. However, in the case of a real valued resistance like here, the angles are equal. We call $\omega t + \phi_U$ the phase of the voltage and $\omega t + \phi_I$ the phase of the current. As in Joule's law, the power output per second equals: $P(t) = U(t)I(t)$.

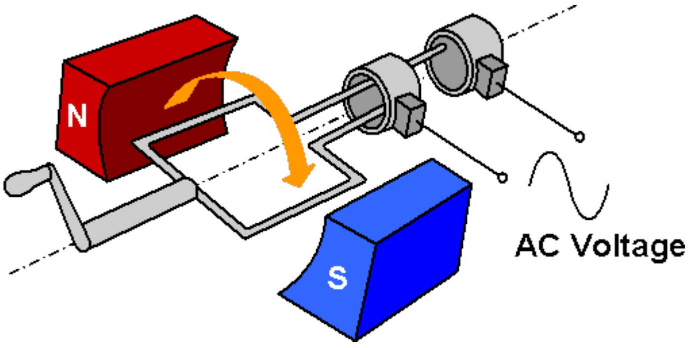


FIGURE 4.1: The generation of alternating voltage and alternating current (AC) by spinning a coil in a magnetic field. source: http://macao.communications.museum/eng/exhibition/secondfloor/MoreInfo/2_4_1_ACGenerator.html

Furthermore, the *period* T and the *frequency* v of both the alternating voltage and the alternating current are given by:

$$T = \frac{2\pi}{\omega} \quad \text{and} \quad v = \frac{1}{T} = \frac{\omega}{2\pi} \ .$$

The period signifies the time between two consecutive moments with the same phase, expressed in seconds. The frequency denotes the number of periods per unit of time. The unit of the frequency is *Hertz*. In general, the period and frequency of voltage and current could be different, but in reality they often turn out to be almost equal.

4.1.2 Complex notation

Now we move on to the complex notation of the voltage and current:

$$U(t) = U_0 \ e^{j(\omega t + \phi_U)} \ \text{ and } \ I(t) = I_0 \ e^{j(\omega t + \phi_I)} \ .$$

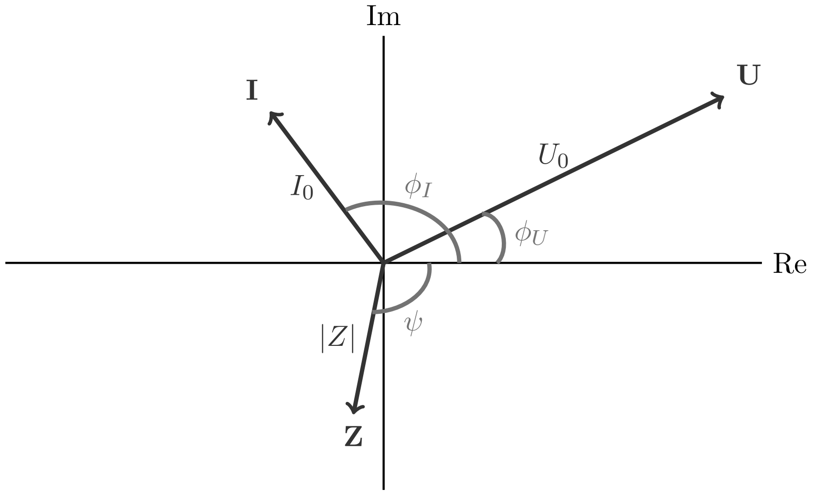
Observe that $\text{Re}(U_0 \ e^{j(\omega t + \phi_U)}) = U_0 \ \cos(\omega t + \phi_U)$, so the complex notation is a well-defined extension of the real time dependent voltage. The same holds for the current. U and I are yet time dependent rotating vectors in the complex plane. The real component is the measurable physical quantity (at a certain point in time). Note that the letter j is used instead of the letter i to represent the complex number $\sqrt{-1}$, as is often done in electrical engineering.

As mentioned before, the complex ratio $\frac{U}{I}$ is called the *impedance*, Z. If there is a phase difference $\psi = \phi_U - \phi_I$, we have:

$$Z \ = \ \frac{U_0}{I_0} \ e^{j\psi} \ = \ |Z| \ \cos \psi \ + \ j|Z| \ \sin \psi, \ \text{ where } \ |Z| = \frac{U_0}{I_0} \ .$$

Note that Z is time invariant. Remember, the real part is called the resistance and the imaginary part the reactance. The former is the actual resistance of the assets in the circuit and expresses the permeability. If $\psi = 0$, the impedance is a real number and corresponds to an ideal resistance. We define the *admittance* to be $\frac{1}{Z}$.

Example 4.1. Sample vectors of the voltage U, the current I and the impedance Z in the complex plane at a certain point in time after a multiple of the period T:



We remark that U and I rotate counter-clockwise over time with the same speed ω, but Z is time invariant and stays stationary in the complex plane. After one period, U and I made precisely one round.

Hitherto we regarded the physical quantities as time dependent objects (apart from the impedance Z). However, it is more practical to work with constant values for those quantities. This can be realized as the waves will be flattened out over a big time scale, so it is acceptable to consider their mean values. To that end, we define the *voltage magnitude* as $U_M = \frac{1}{2}\sqrt{2}\ U_0$ and the *current magnitude* as $I_M = \frac{1}{2}\sqrt{2}\ I_0$. We reduce the complex voltage and current to $U = U_M\ e^{j\phi_U}$ and $I = I_M\ e^{j\phi_I}$, respectively. Remark that this does not change the impedance $Z = \frac{U}{I}$.

Furthermore, the power can be seen as a constant complex vector. We denote S to be the complex power, also called the *apparent* power, defined by:

$$S = U\ \overline{I} = U_M\ I_M\ e^{j\psi} = P + jQ,$$

where $\overline{I}$ is the complex conjugate of I. One can understand this equation as the complex variant of Joule's law. P is called the *active* power, this is the real power as mentioned before. Van der Meulen (2015) [20] showed that the average real power equals $\frac{U_0 I_0}{2}$ $\cos(\psi)$. Since this expression equals $U_M\ I_M\ \cos(\psi)$, we conclude that the real component of S equals P. The complex component Q comprises the *reactive* power. From now on, we only use the constant values of the physical quantities, which we assume to be complex numbers.

In the next section, we explain the load flow model. This model computes the values of the physical quantities in an electricity distribution grid. The load flow model combines graph theory, linear algebra and electromagnetism.

4.2 Load flow model

Electric power flow is a nonlinear quantity as it is the solution to a set of nonlinear equations. The problem of how to solve these nonlinear equations is called the *load flow problem*. We will discuss the *load flow equations* and the associated problem after we have treated some definitions. We use the lecture notes *Introduction To Load Flow* by Kirtley (2011) [21].

4.2.1 Load flow problem

In an electricity distribution network, current flows through the cables, induced by the voltage differences in the nodes. This power flow is determined by the voltage at each node and the impedances of the cables. These values allow us to compute the current in the cables. The power flow into and out of a node is the sum of the power flows of all cables connected to the node. We depict the power flow in node i by S_i. The load flow problem consists of finding the voltage magnitude U_{Mi} and voltage angle ϕ_{Ui} in each node. We assume the impedances of the cables are known in advance.

Recall the definition of the $n \times m$ incidence matrix I_G of a graph G as given in definition 3.1 in section 3.1. Then we can define two *admittance matrices*:

Definition 4.2. Let G be a network graph with nodes $\{v_1, \ldots, v_n\}$ and cables $\{e_1, \ldots, e_m\}$ and let Z_i be the impedance of cable e_i. We define the $m \times m$ *edge admittance matrix* A_G by:

$$(A_G)_{i,j} = \begin{cases} \frac{1}{Z_i} & \text{if } i = j \\ 0 & \text{else} \end{cases}$$

The $n \times n$ *node admittance matrix* Y_G is defined by: $Y_G = I_G \cdot A_G \cdot {I_G}^T$.

Notice that the edge admittance matrix A_G is nothing more than the admittance of each cable in the right place on the diagonal. The node admittance matrix Y_G can be seen as an extension of the Laplacian matrix such that the admittances of the cables are included, as we have for the Laplacian matrix: $L_G = I_G \cdot {I_G}^T$. As a result, we can show in a similar way to the proof of $L_G = I_G \cdot {I_G}^T$ in lemma 3.5 in section 3.1:

- $(Y_G)_{i,i}$ equals the sum of all admittances of the edges connected to node v_i.
  Compare: $(L_G)_{i,i} = d(v_i)$.
- If $i \neq j$, then $(Y_G)_{i,j}$ equals *minus* the sum of the admittances of all edges connected *directly* between nodes v_i and v_j. (In a simple graph this is at most the admittance of one edge.) As a consequence, Y_G is symmetrical.
  Compare: if $i \neq j$, then $(L_G)_{i,j}$ equals minus the number of edges connected directly between nodes v_i and v_j.

We conclude from these observations that the node admittance matrix Y_G represents the sum of the admittances of the cables connected to a certain node (on the diagonal) and the sum of the admittances of the cables between two nodes (not on the diagonal). The name of the matrix is therefore well chosen.

The current I_i of the node v_i is the sum of the currents in the cables connected to v_i *flowing away from* v_i, so we multiply a current with -1 if it flows into v_i. Let I_N be the vector of complex node currents and let U_N represent the vector of complex node voltages. Then we have: $I_N = Y_G \cdot U_N$, as a multivariate variant of Ohm's law. Moreover, we have for each individual node: $I_i = \sum_{j=1}^{n} (Y_G)_{i,j}\, U_j$. Hereby the complex power flow at node i equals:

$$S_i = U_i\, \overline{I_i} = U_i \sum_{j=1}^{n} \overline{(Y_G)_{i,j}}\; \overline{U_j}\ . \tag{4.1}$$

The load flow problem consists of solving these n *load flow equations*, given nodes with different constraints. At each node, six different quantities could be specified: voltage magnitude and angle, current magnitude and angle, real and reactive power. To set up the load flow problem, we need to know two of these six quantities for each node. Typical constraints for different types of nodes (buses) are:

- **Generator bus**: real power P and voltage magnitude U_M
- **Load bus**: real power P and reactive power Q
- **Slack bus**: voltage magnitude U_M and voltage angle ϕ_U

The type 'Slack bus' is a voltage source, in our case an HV/MV transformer. This node does not have real nor reactive power constraints.

To sum up, the load flow problem consists of computing the node admittance matrix Y_G from the graph G and the m impedances of the cables, assumed to be known. Furthermore, the values of two quantities at each node are given and we need to found the solutions to the equations (4.1). Since n is large in an electricity distribution network, equations (4.1) form an extensive system of nonlinear equations. Customarily, we approximately solve (4.1) using the Newton-Raphson iterative technique. The next subsection explains how this method is applied to the load flow problem.

4.2.2 Solving the load flow equations

Since n is large and the equations are nonlinear, solving equations (4.1) exactly is generally too complex and takes too much time. To speed up the calculations, most commonly the solutions are approximated using an iterative technique. The Newton-Raphson method is used most frequently. First, we argue how one finds the cable currents given the node voltages. Hereto, it suffices to focus on finding the node voltages in the first place. Second, we will explain the Newton-Raphson method applied to the load flow equations.

As described in the previous subsection, we have one slack bus without power constraints, but with given voltage magnitude and angle. On top, we have generator buses and load buses. We know the real and reactive power at each load bus and the real power and voltage magnitude at each generator bus. We would like to know the voltage magnitudes and angles at all nodes, because these allow us to check whether the voltage capacities have been exceeded. Moreover, we can then calculate the current magnitudes and angles in the cables using the impedances and the incidence matrix:

$$I_E = A_G \cdot {I_G}^T \cdot U_N \ , \tag{4.2}$$

where I_E is the vector of the complex cable currents. This is because the complex currents in the nodes are given by $I_N = Y_G \cdot U_N$ and we convert the cable currents to node currents by $I_N = I_G \cdot I_E$. Combining this gives: $I_G \cdot A_G \cdot {I_G}^T \cdot U_N = I_G \cdot I_E$. Although not necessarily $I_E = A_G \cdot {I_G}^T \cdot U_N$ in this case (as I_G possibly has no left-inverse), of course equality holds if we take I_E to be $A_G \cdot {I_G}^T \cdot U_N$.

We give another argument why equation (4.2) holds. We could find a cable current by calculating the complex voltage difference between the ends of the cable (we already computed the node voltages) and then using $I = \frac{\Delta U}{Z}$, as we know the impedances. This results in the same current per cable as calculating $I_E = A_G \cdot {I_G}^T \cdot U_N$, for ${I_G}^T \cdot U_N$ actually computes the voltage difference for each cable (check this) and multiplication with A_G is the same as multiplication with $\frac{1}{Z_i}$ per cable.

To conclude, given the voltage magnitudes and angles of the nodes, we find the current magnitudes and angles of the cables in two straightforward matrix multiplications. Hence we can check the current capacities.

We now formulate the Newton-Raphson iteration applied to the load flow equations. We follow the lecture notes *Modelling and Analysis of Electric Power Systems* by Andersson (2008) [18], sections 6.2 and 6.3.

We need to know the voltage angles of the generator buses and the voltage magnitudes and angles of the load buses. Suppose $\{v_2, \ldots, v_g\}$ are the generator buses and $\{v_{g+1}, \ldots, v_n\}$ are the load buses (v_1 is the slack bus). Let X be the vector of unknown voltage angles and magnitudes (in order of the node numbers) and let f be a function of X that depicts the difference between the solution of equations (4.1) applied to X and the given known real and reactive powers:

$$X = \begin{pmatrix} \phi_{U\,2} \\ \vdots \\ \phi_{U\,n} \\ U_{M\,g+1} \\ \vdots \\ U_{M\,n} \end{pmatrix} := \begin{pmatrix} \phi_U \\ U_M \end{pmatrix} \quad \text{and} \quad f(X) = \begin{pmatrix} P_2(X) - P_2 \\ \vdots \\ P_n(X) - P_n \\ Q_{g+1}(X) - Q_{g+1} \\ \vdots \\ Q_n(X) - Q_n \end{pmatrix} := \begin{pmatrix} P(X) - P \\ Q(X) - Q \end{pmatrix}$$

Notice that $P_j(X)$ denotes the active power flow out of node v_j given by equations (4.1) provided certain 'guessed' values of X. P_j denotes the measured value of the active power in node v_j. Similar meanings hold for $Q_j(X)$ and Q_j with regard to the reactive power flow out of node v_j. $P(X) - P$ are all active power mismatches, $Q(X) - Q$ all reactive power mismatches. The load flow equation can now be written as:

$$f(X) = \begin{pmatrix} P(X) - P \\ Q(X) - Q \end{pmatrix} = 0$$

Of course, finding the values of X without power mismatches means finding the solution.

As the Newton-Raphson technique only approximates the solution, we have to define an acceptable error $\epsilon > 0$ beforehand. Let X_0 be an initial guess of X, this is the starting value, and let the iteration counter $p = 0$. The Newton-Raphson technique creates a sequence $X_1, X_2, \ldots$ of improved values of X, until at some iteration z values are found such that $|f_i(X_z)| \leq \epsilon$. In that case, we have an approximate solution. We give an outline of the Newton-Raphson algorithm:

1. Compute $f(X_p)$.
2. Test the convergence: if $|f_i(X_p)| \leq \epsilon$ for all i, then X_p is the approximate solution: stop.
3. Compute the Jacobian matrix $J(X_p)$.
4. Update the solution: $\Delta X_p = -J^{-1}(X_p)\ f(X_p)$, so $X_{p+1} = X_p + \Delta X_p$.
5. Update the iteration counter: $p \to p+1$ and go to step 1.

Some clarifications to the steps of the Newton-Raphson algorithm:

- The Jacobian matrix J in this case equals:

$$J = \begin{pmatrix} \frac{\partial P(X)}{\partial \phi_U} & \frac{\partial P(X)}{\partial U_M} \\ \frac{\partial Q(X)}{\partial \phi_U} & \frac{\partial Q(X)}{\partial U_M} \end{pmatrix}$$

- The iterative updates to the solutions $\Delta X_p = X_{p+1} - X_p$ are determined from the equation:

$$\begin{pmatrix} P(X_p) - P \\ Q(X_p) - Q \end{pmatrix} + J(X_p)\Delta X_p = 0$$

  This is based on the Taylor expansion:

$$f(X_p + \Delta X_p) \approx f(X_p) + J(X_p)\Delta X_p$$

It should be noted that the Newton-Raphson iteration does not always converge, but in the case applied to the load flow equations it generally does.

Instead of computing the Jacobian in each iteration, we could opt for a constant Jacobian: $J(X_p) = J(X_0)$. This increases the number of iterations needed for convergence, but the computation burden for each iteration is lower. As the overall performance may be better in this way, it is worth considering.

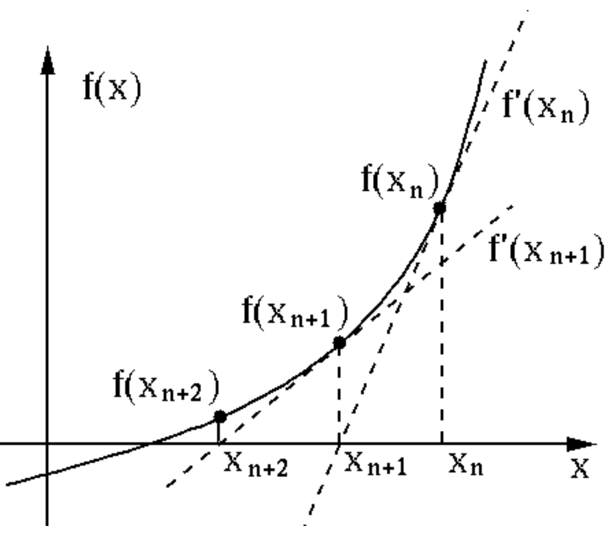


FIGURE 4.2: Sketch of the Newton-Raphson iterative technique (univariate case).

4.3 Linear model

Although electric power flow is a nonlinear quantity, it can be studied as if it were linear, thereby avoiding the calculation of multiple iterations as in the Newton-Raphson method. This makes solving the equations computationally less expensive and therefore yields this model applicable to a large scale distribution grid. Werner van Westering developed this so called *linear model* recently at Alliander. He models the load buses and generator buses as 'constant impedances' instead of buses with constant power demands. Therefore, one needs to convert the power use into an equivalent resistance. To realize this idea, Van Westering adds to each node (apart from the slack bus) a virtual cable connected to the ground. The power demand of the node is then translated to a resistance of the additional cable. This is done by combining Ohm's law and Joule's law:

$$R_i = \frac{{U_{ref}}^2}{P_i}. \tag{4.3}$$

Here R_i is the equivalent resistance of the additional cable at node v_i and P_i is the known real power of the node v_i. U_{ref} is an assumed reference voltage, as the voltage at the node is not known (otherwise there is no load flow problem). Hereby the solution will have an error, but can nevertheless be a good approximation. The network now consists of a voltage source (slack bus), resistors and ground connections. The ground connections are additional nodes (resulting from the additional cables) and have a voltage of 0 Volt. Note that $U_{ref} = \Delta U$ in this case, consistent with Ohm's law and Joule's law. We still use the equation:

$$I_N = Y_G \cdot U_N, \text{ where } Y_G = I_G \cdot A_G \cdot {I_G}^T \ , \tag{4.4}$$

as in the previous section. However, the vectors and matrices are extended with the $n-1$ additional nodes and $n-1$ additional cables at the beginning, i.e. the vectors I_N and U_N have the additional nodes as first nodes. A_G still consists of all cable admittances on the diagonal, where the first $n-1$ elements on the diagonal are the resistances R_i of the additional cables. The first $n-1$ columns of I_G correspond to the additional cables and the first $n-1$ rows of I_G correspond to the additional nodes. We assume all cable resistances (including the new ones) are known, as well as the structure of the graph, so we can calculate A_G, I_G and then Y_G.

We want to solve equation (4.4), but we neither know vector I_N nor U_N completely, hence we cannot solve this equation directly. For this reason, we split the equation in two separate equations that can be solved. We partition the rows of I_N, Y_G and U_N such that the segments I_2 and U_1 are defined:

$$I_N = \begin{pmatrix} I_1 \\ I_2 \end{pmatrix}, \quad Y_G = \begin{pmatrix} K & L \\ L^T & M \end{pmatrix} \quad \text{and} \quad U_N = \begin{pmatrix} U_1 \\ U_2 \end{pmatrix}.$$

Then I_2 consists of the currents in the load buses and generator buses, as no power enters or leaves the network in these buses, so $I_2 = 0$. U_1 consists of the voltages at the new ground connections and the voltage at the slack bus, as these values are known. Note that this is a well-defined partition, as I_1 and U_1 concern the new ground connections and the slack bus, and I_2 and U_2 concern the load buses and generator buses. This forms an actual splitting of the nodes. We now have, using equation (4.4) and the segmenting:

$$0 = L^T\, U_1 + M\, U_2\ , \quad \text{or:} \quad M\, U_2 = -L^T\, U_1$$

Only U_2 is not known in this equation. We prefer to avoid the computation of a large matrix inverse, in this case M^{-1}, but we could calculate U_2 with a (sparse) QR decomposition method, for example.

After this computation, we know all node voltages U_N. The cable currents can be found in the same way as before by: $I_E = A_G \cdot {I_G}^T \cdot U_N$.

We present some considerations made by Van Westering with regard to the accuracy of his model:

- If the power demand of some node is almost zero, then the equivalent resistance tends to infinity by equation (4.3). This is undesirable and prone to computational errors. However, we can avoid this problem by letting the power consumption be at least a few Watts. This has no big influences on the calculation.
- As may be noted, the model only uses *real* powers and *resistances* (instead of *impedances*). Therefore, the model neglects the reactive currents. We could solve a separate identical problem to find the reactive currents and afterwards combine the results. In practice, however, the reactance is at least an order of magnitude lower than its resistance. Similar results hold for the reactive power. To simplify the calculations, the reactive components are omitted.

4.4 Load flow in the $m - 1$ problem

As announced in chapter 1, we have to check that a possible reconfiguration of the MV network does not have current or voltage values that exceed the capacities. In chapter 2 and 3 we only focused on solving the mathematical part of the problem, but now we need to add the physical boundary conditions. This involves making some assumptions on the known values of the physical quantities (something we can never be completely sure of as we do not know all these values in reality). On the other hand, we need to take the voltage and current capacities into account, which includes the calculation of the load flow.

We take the same assumptions as outlined in section 4.2 about the load flow model. This means that we assume the impedances of the cables are given beforehand as well as two quantities per node typical for the kind of node: slack bus, load bus or generator bus. This possibly involves complex numbers.

As a reconfiguration changes the way the power flows through the network, it probably changes the values of the physical quantities in the network. Some nodes will have a higher voltage than before, other cables a lower current, for example. Of course, the voltages and currents are not allowed to have all values in practice, as this could overheat some assets in the network and can cause dangerous situations. Therefore, we have certain boundary conditions for each node and each cable, called the *capacity conditions*:

- Each node v_i has its own voltage boundaries: lower limit U_i^{min} and upper limit U_i^{max}. This means its voltage U_i in a certain configuration should be between the bounds: $U_i^{min} \leq U_i \leq U_i^{max}$.
- Each cable e_i has its own current boundary: upper limit I_i^{max}. This means its current I_i in a certain configuration should be under the upper limit: $I_i \leq I_i^{max}$.

We assume the node voltages U_i and cable currents I_i can be calculated either by the load flow model or the linear model. As we are searching for reconfigurations of the MV network in the $m-1$ problem, we need to check the voltage and current capacities each time we try a new reconfiguration. Thus, we use the physical model for each possible reconfiguration in combination with the predetermined assumed values of some quantities. If one of the capacities is exceeded in the reconfiguration, the algorithms (in R) search for another reconfiguration. Otherwise, the results of the reconfiguration are used: a possible switch-over may be found for certain 'broken' cables.

As the linear model calculates the values much faster than the load flow model, it is possibly more suitable to use over and over again to check possible reconfigurations. Therefore, we ultimately decided to only incorporate this model in the implementation. For completeness, we explained the classical load flow model here too.

Chapter 5

More Graph-theoretical Tools

Before we can explain the practical implementation of our solution to the $m-1$ problem in the program R, we need some more mathematical tools to handle the problem. This chapter addresses these graph-theoretical algorithms and methods. In the next chapter, chapter 6, we show the relevance of these tools in practice, although for some means the utility will already be clear.

We start with the presentation of an algorithm to find all bridges of a graph, these are the graph-theoretically unswitchable edges. Second, we introduce a solution to generalize the eventual algorithm over the whole MV network at once, instead of restricting the network to an MV area having only one HV/MV transformer. Last, we discuss how to anticipate a problem concerning the initial state of the network. In fact, the initial configuration is not necessarily a spanning tree.

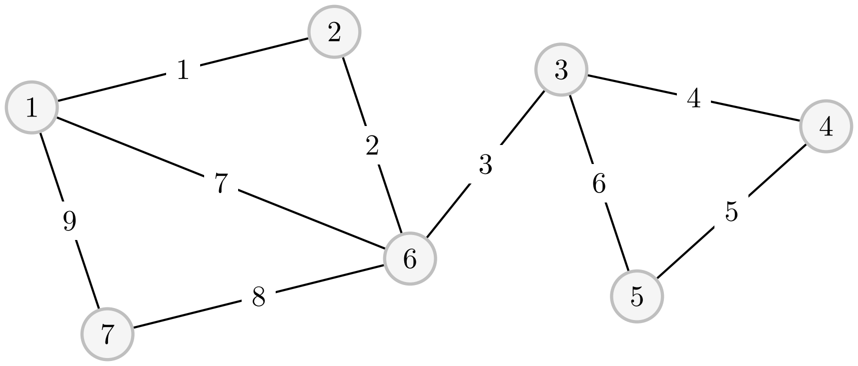
5.1 An algorithm to find all bridges

In the practical algorithm in R, we try to match spanning trees with multiple edges to discover switching solutions, as explained in section 2.3. We will continue this process until we found a switchover for each edge, in case it exists. For this reason, it is useful to know the edges that cannot be switched beforehand. With respect to the physical condition, we could not determine this in advance. However, we could compute the graph-theoretically unswitchable edges. This could save us computation time in the overall algorithm, as we do not go on to search for switchovers that do not exist.

Recall the definition of a bridge (in a connected undirected graph G): an edge that disconnects the graph upon removal. For the sake of clarity, we mean bridges of the entire MV network, not of some configuration which is a spanning tree. In a spanning tree, each edge is a bridge. We are thus interested in finding the bridges of the entire graph, i.e. the edges that are unswitchable at all, even without the computation of the physical values. Note that such edges ensure that the network does not satisfy the $m-1$ principle, but we want to know to what extend it does, as we noted before.

Before we treat an algorithm to find all bridges, note that we have found special kinds of bridges already by the Andrei-Chicco reduction, namely the edges in branches.

The first step in this reduction deletes the edges in the 'network outflows', as these edges do not have a switchover. These edges in branches are bridges (and we have already filtered them out), but there could be more bridges. For example edge 3 in the following graph is a bridge, but it is not in a branch:



Multiple articles present algorithms to find all bridges of a graph. Even a lot of linear-time algorithms have been given, meaning that the time complexity of these algorithms is just $O(n+m)$. The algorithms use different graph structures, like 'ear decompositions' or 'low-points' (developed by Tarjan (1972) [13]), the latter is used a lot. However, 'low-points' do not give the most natural and simple solution. Gabow (2000) [14] presents an easier algorithm using path-generating rules instead of low-points. We call this style of algorithms *path-based*. Schmidt (2012) [12] presents in his lecture notes for the course *Advanced Graph Algorithms*, given at the Max Planck Institute for Informatics in Saarbrücken, an even simpler linear-time algorithm that finds all bridges of a graph. We implement Schmidt's method because of its simplicity and time complexity. Schmidt's approach is path-based too and uses no low-points. It is however related to 'ear decompositions', which we do not treat here.

5.1.1 The algorithm by Schmidt

Before we present the entire algorithm by Schmidt (2012) [12], we discuss its approach and treat some necessary terminology.

Let G be a given connected undirected graph in which we want to determine the bridges. G could have parallel edges and we assume $n \geq 3$. Furthermore, loops do not matter in the algorithm. We will decompose G in sets of paths and cycles, both paths and cycles will be called a **chain**.

The approach

The first step in the algorithm will be a depth-first search on G, as described in section 2.2. A more detailed explanation of a depth-first search can be found in Bondy and Murty (2008) [1], section 6.1. We can choose a root r at random in advance. Then the depth-first search on G leads to a DFS-tree T rooted at r, as G is connected. Besides, we let the depth-first search assign a **depth-first index** (DFI) to every vertex. This index expresses when a vertex is added to the tree T in the formation of T: if vertex v is the i-th visited vertex, the DFI of v is i. The DFI is a preorder, as defined in subsection 3.4.3.

The second step in the algorithm is to orient all edges in T towards r in the graph G. We refer to these edges as **forward edges**. Thirdly, we orient the other edges in G away from r and we call these remaining edges **back edges**. Thus, we have made the undirected graph G directed by defining an orientation for each edge based on the DFS-tree. Each back edge e lies in precisely one directed cycle $C(e)$ containing only e as back edge. We will prove this important property in the next subsection.

The next step will decompose G into chains, by applying the procedure explained below, also depicted in lines 5-8 of the pseudo algorithm on the next page. Initially, we mark each vertex and each edge as *unvisited* (in line 4), but this changes during the execution of the procedure.

We explain the method in a nutshell. We create chains beginning in back edges e that start in turn in certain vertices v. Here we consider the vertices (the starting points of the back edges) in the DFI order. The chains are the beginning of directed cycles $C(e)$. By the announced lemma in the next subsection, these cycles $C(e)$ exist and are unique. In the creation of the chains, we pass through the $C(e)$ until we visit a vertex that we visited before. Meanwhile, we mark the traversed vertices and edges as visited. This completes the computation of the chains. The edges that we did not visit in the computation are precisely all bridges. We will prove this in the next subsection.

Thus, a chain, a partial traversal of an $C(e)$, stops at the latest at v and forms either a directed path or a directed cycle. We represent a chain by the array of edges in the order they were visited. We call the i-th chain found in this procedure C_i. Note that each back edge is traversed at least once and hereby included in a chain, as we always start traversing a back edge, even if v was visited before. Only after traversing e we explore whether we should further traverse $C(e)$ or not. By the definition of lines 5 and 6, we ensure that each back edge is only traversed once. We traverse the forward edges at most once, as after covering such an edge the starting point of the edge is marked visited.

There are $m - n + 1$ chains, as each back edge (each edge not in T) creates exactly one chain. We refer to the set $C = \{C_1, \ldots, C_{m-n+1}\}$ as a **chain decomposition**.

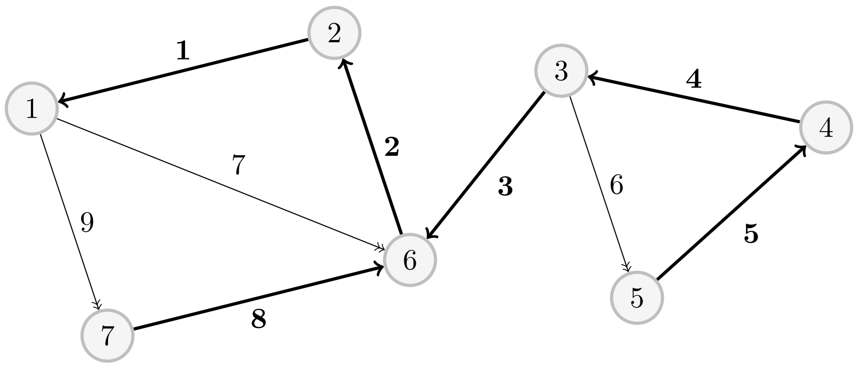
The algorithm

We now present an overview of the algorithm by Schmidt (2012) [12]. First, we compute a DFS-tree and orient the edges of G. This enables the computation a chain decomposition in lines 5-8. After this computation, we know which edges are bridges, namely those that were not visited in the computation of the chains.

```
BRIDGES: function to find all bridges of a connected undirected graph $G$
Input arguments: $G$ (given by its edges), root $r$

1. compute a DFS-tree $T$ rooted at $r$ of $G$, save the DFI of the vertices
2. orient all edges in $T$ towards $r$ in $G$, these are the forward edges
3. orient all other edges in $G$ away from $r$, these are the back edges
4. mark all vertices and edges as unvisited
5. for each vertex $v$ in the order of the DFI
6.     for each back edge $e$ starting in $v$
7.         start in $e$, traverse $C(e)$ until we visit a vertex that is marked visited
8.         meanwhile, we mark every vertex and edge that we visit as visited
9. output(all edges that are marked unvisited)
```

Example 5.1. Below we show the execution of BRIDGES on the graph drawn on page 53, where we take vertex 1 as root. The bold arrows form the DFS-tree, computed in line 1 of the pseudo algorithm. Lines 2 and 3 make the graph directed. The edges in the DFS-tree are the forward edges, the remaining edges are the back edges.



The order in which the nodes are added to the DFS-tree is: 1, 2, 6, 3, 4, 5, 7. This determines the depth-first index too. For example, vertex 6 is added as third vertex to the tree, so the DFI of vertex 6 equals 3. We need the DFI in line 5 in the pseudo code. The vertices in the DFI order in turn decide the order in which we consider the back edges (in line 6). If there are multiple back edges starting in some vertex v, we consider them in the natural order.

After executing lines 5-8, we see the chain decomposition of the example graph is $C =$

$\{C_1,\ C_2,\ C_3\}$, where:

$$C_1 = \{7,\ 2,\ 1\}$$
$$C_2 = \{9,\ 8\}$$
$$C_3 = \{6,\ 5,\ 4\}$$

Note that the back edges are the starting edges of the chains. We now know the set of bridges of the graph is $\{3\}$, as 3 is the only edge not contained in any chain in the chain decomposition.

Before we proof the correctness of BRIDGES in the next subsection, we first note the difference between this algorithm by Schmidt and the bridge test in the algorithm by Gabow and Myers (subsection 3.4.1). In the algorithm by Gabow and Myers, we need to know whether a specific edge in a (present) graph is a bridge. Therefore, we define the bridge test, which uses a property of descendants in a graph. It would be inefficient to use this bridge test to find all bridges in a given graph, as we need to perform this test on all edges. The bridge test uses $O(n)$ time, so the time complexity of this way to find all bridges is not linear but $O(n \cdot m)$. (If we save some intermediate results, we could only speed it up to $O(n^2)$ time. In the worst case, we need to compare values $P(v)$, $P(w)$ and $H(v)$ for all combinations of different vertices v and w, which takes $O(n^2)$ time.)

On the other hand, we could not use this algorithm by Schmidt to check whether an edge is a bridge in the algorithm by Gabow and Myers, as BRIDGES is designed to find *all* bridges. This is not efficient in the algorithm by Gabow and Myers, as we only need to know whether a particular edge is a bridge. Then the bridge test uses $O(n+m)$ time instead of just $O(n)$.

To conclude, the difference between the bridge test in the algorithm by Gabow and Myers and the algorithm to find all bridges here is too big to combine these algorithms efficiently. It is clear that we do need some of the same properties of a graph in both algorithms (like bridges, descendants and preorders).

5.1.2 Proofs: Correctness of the algorithm

This subsection shows the correctness of the algorithm presented. First, we prove the following lemma concerning directed cycles $C(e)$, as promised in the previous subsection. In the proof, we represent a path by its edges.

Lemma 5.2. *Let e be a back edge in the connected graph G, the graph made directed using a DFS-tree T as described in* "The approach" *on page 72. Then e lies in precisely one directed cycle $C(e)$ containing only e as back edge.*

Proof. Let $e = (u, v)$ be a back edge. We first show there is at most one cycle $C(e)$. Suppose there are two different directed cycles C_1 and C_2 containing only e as back edge. Then $C_1 \setminus \{e\}$ and $C_2 \setminus \{e\}$ form two different directed paths from v to u in T, as these paths contain no back edges. However, as the spanning tree T contains exactly one (undirected) path from v to u, this leads to a contradiction. To conclude, there is at most one cycle $C(e)$.

Now we just need to prove that there is at least one such cycle $C(e)$. We claim that there is a directed path P from v to u in T. We will proof this claim in a minute. Note that e forms a directed path from u to v on its own and e does not lie in T, so if P exists, $C = P \cup \{e\}$ is a directed cycle containing only e as back edge. Thus, the claim shows there is at least one directed cycle $C(e)$ containing only e as back edge.

There is an undirected path from v to u in the spanning tree T. However, it is not obvious that this path, henceforth called P, is a *directed* path if $u \neq r$. (If $u = r$, P is directed by the direction of the edges in T.) Step by step, we will nonetheless show that P is directed.

As the spanning tree T is oriented towards r, there is a unique directed path U from u to r and a unique directed path V from v to r. One of the following two cases is true: (1) u lies on the path V, or (2) there is a vertex $w \neq u$ on both U and V such that the unique path W from w to r equals $U \cap V$. (Note that we cannot divide $U \cap V$ in two unconnected parts, as we could identify a cycle in T in that case. Further, note that $w = r$ might be the case.) If (1), then $V \setminus U$ is a directed path from v to u and we are done. If (2), then $V \setminus W$ is a directed path from v to w, $U \setminus W$ is a directed path from u to w and these paths do not intersect. The unique path P must be the combination of these two paths, but then P is not directed properly. We prove that case (2) leads to a contradiction.

First note that the DFI of w is lower than the DFI of both u and v. Without loss of generality, the DFI of v is lower than the DFI of u (the other case is analogue). At the points in the formation of T where v is under view but u is not yet added, we did not append e to T. u is eventually added via a path starting in w, this path does not intersect with the path from v to w (that would form a cycle). As we prefer to add the most depth-first edge and the first edge in the path from w to u is less depth-first than e, then we could not add e when v was under view for the last time (or earlier), otherwise we definitely did. Therefore, e forms a cycle in T at that time already (when v is under view but u is not yet added). Then there must be a path from v to u in T at that time. However, u is not yet added so this cannot be the case: a contradiction. Thus, case (2) leads to a contradiction and (1) is always true. Thus, P is a properly directed path from v to u. □

We just need one more lemma before we are able to prove the correctness of the algorithm. This lemma is probably well-known (see section 1.4 in the book *Graph Theory* by Diestel (2010) [15], for example):

Lemma 5.3. *Let G be a connected undirected graph. An edge is a bridge in G iff it is not contained in any cycle.*

Proof. Let e be an edge of a connected graph G.
Suppose $e = (u, v)$ is a bridge. Then there is no path from u to v in $G \setminus \{e\}$, as e could

be removed without disconnecting G otherwise. If e were in any cycle C, then $C \setminus \{e\}$ would be a path from u to v. Concluding, e is not contained in any cycle.
Suppose now $e = (u, v)$ is not contained in any cycle. Then there is no path from u to v without e, for e would lie in a cycle otherwise. The only path from u to v is then the edge e on its own. Hence in $G \setminus \{e\}$ there is no path from u to v, so $G \setminus \{e\}$ is unconnected by proposition 2.6. Thus, e is a bridge. □

We may assume the algorithm forms a chain decomposition in the right manner. Then the following theorem expresses the correctness of the algorithm by Schmidt.

We will use the following notation in the proof of the theorem: if T is a tree rooted at r and vertex $x \in T$, then we denote by $T(x)$ the subtree of T containing x and all descendants of x (independent of the edge orientations of T).

Theorem 5.4. *Let C be a chain decomposition of a connected graph G. Then an edge e in G is a bridge iff e is not contained in any chain in C.*

Proof. Suppose e is a bridge. Note that we form the chains from directed cycles of type $C(a)$ where a is the only back edge in the cycle, as stated in the pseudo algorithm and lemma 5.2. As a consequence, if e were contained in a chain in C, then e is part of a particular cycle $C(b)$ where b is the back edge starting the chain. This cannot be the case according to lemma 5.3, so e is not contained in any chain in C.

Now suppose e is an edge that is not contained in any chain in C. Note that e cannot be a back edge as by the definition of the chain decomposition, each back edge is contained in a chain. Thus, e is a forward edge and it is contained in the DFS-tree T used for computing C. Let T be rooted at r and let $e = (u, v)$. As e is directed towards r, $T(u)$ does not contain e and we have $T(u) \neq T$. Furthermore, we will show there is no back edge with exactly one end in $T(u)$:

Suppose there is at least one back edge with exactly one end in $T(u)$. As back edges are directed away from r, its end point should be in $T(u)$. Suppose $b_1, \ldots, b_p$ are such back edges. We first show that these back edges are traversed before the back edges that lie entirely in $T(u)$ in the formation of the chain decomposition:

Suppose for the contrary that back edge $c = (c_l, c_r)$ lies entirely in $T(u)$, back edge $b = (b_l, b_r)$ only has its end point in $T(u)$ and c is traversed before b in the creation of the chain decomposition. Then c_l has a lower DFI than b_l, because clearly $c_l \neq b_l$. In addition, all vertices in $T(u)$ have a lower DFI than b_l by the property of a preorder, given in lemma 3.25. However, then we could add b to $T(u)$ in the formation of T at a moment that we consider edges starting in $T(u)$ (as composed so far), unless b would form a cycle in T created so far. As b_l is not yet used in T, b does not form a cycle in the tree so far. Thus, we should add b to T while considering vertices in $T(u)$, but we did not do this. This contradicts the assumption that T is a DFS-tree. Therefore, b is traversed before c. In this way, we can prove that all back edges $b_1, \ldots, b_p$ are traversed before all back edges entirely in $T(u)$.

Without loss of generality, assume now that b_1 is the first edge of all $b_1, \dots, b_p$ that is traversed in the making of the chain decomposition. We conclude that $e \in C(b_1)$, for $C(b_1)$ exists and b_1 could never form a directed cycle without e. As none of the vertices in $T(u)$ have been traversed before (as none of the back edges towards $T(u)$ have been yet, just as none of the back edges inside $T(u)$), we can traverse $C(b_1)$ at least up to v, as none of the vertices inside $T(u)$ have been visited. Thus, assuming there are back edges $b_1, \dots, b_p$, then e is in a chain, which contradicts the assumption.

We go back to the proof of the lemma itself. We now have that $e = (u, v)$ is a forward edge and there is no back edge with exactly one end in $T(u)$. Removal of e from G then results in a disconnected graph, as there is no edge that connects a vertex in $T(u)$ with a vertex in $T \setminus T(u)$ ($\neq \emptyset$). Since there is only one such forward edge (by the definition of a tree and its descendants) but that one is removed (e is removed) and we proved there is no such back edge. Hence, e is a bridge. □

5.1.3 Proofs: Complexity of the algorithm

Lastly, we prove the complexity of the algorithm by Schmidt:

Theorem 5.5. *Let G be an undirected connected graph having n vertices and m edges. All bridges of G can be found by* BRIDGES *in time $O(n + m)$.*

Proof. We will show that all elements of the pseudo algorithm need at most $O(n + m)$ time. First note that lines 1-3, 4-8 and 9 are executed completely sequentially, so the time complexity of BRIDGES is the sum of the time complexities of those individual parts.

We begin with the time complexity of lines 1-3. To compute a DFS-tree T, we need to traverse all vertices precisely once and we need to keep track of $\partial(T)$ in the creation of T. This means we need to add and remove edges to $\partial(T)$ each time we add a new vertex and edge to T. Fortunately, this is bounded in the following sense: if an edge e is removed from $\partial(T)$, it will never be added again to $\partial(T)$. We remove e in two cases: if e is added to T or if some other edge is added to T whereby both ends of e lie in T. In both cases, both ends of e lie in T for the rest of the computation of T. Therefore, e will never be in $\partial(T)$ again, as $\partial(T)$ consists of the edges having precisely one end in T. Thus, each edge will be added to $\partial(T)$ at most once and will be removed from $\partial(T)$ at most once. Furthermore, we add precisely $n - 1$ edges to T and never remove one from T. Thus, the total time to compute T is $O(n + m)$, as expanding T takes $O(n)$ and keeping track of $\partial(T)$ takes $O(m)$. Besides, we save the DFI of the vertices. The creation of the DFI takes $O(n)$ time, as we could see the DFI as an array of vertices in the order in which they were added to T. To orient the edges in G clearly takes $O(m)$ time. Thus, lines 1-3 take $O(n + m)$ time.

Now consider lines 4-8. Line 4 takes clearly at most $O(n + m)$. Lines 5 and 6 consist of a nested for loop, usually taking a lot of time, but this is not the case here. As there are $m - n + 1$ back edges and each back edge (in line 6) will be examined only once (when its starting point is considered in line 5), the time purely needed in lines 5 and 6 is $O(n+m)$. We need to check each vertex once and further need to do something for every back edge (but not for each combination of a vertex and a back edge, fortunately). The time spent in lines 7-8 in total (after the whole execution of lines 5-8) is also $O(n+m)$, as each edge is considered in at most one chain and the same holds for each vertex. We conclude that the total time spent in lines 5-8 is $O(n+m)$ too.

The time needed in line 9 is clearly $O(m)$, as we only need to check which edges were visited, but this has already been computed. Last, note that the space required for BRIDGES is also no more than $O(n+m)$, as G, T, $\partial(T)$, the DFI and the arrays of forward edges, back edges, visited nodes, visited edges and bridges are all bounded by space up to the size of $n+m$. Thus, the total time needed for the execution of BRIDGES is $O(n+m)$. □

The time complexity of the algorithm does not have to be optimal, for theoretically there could be an algorithm enumerating all bridges in $O(m)$. Note that the output of the bridges takes $O(m)$ time, so this is a lower bound. However, in connected graphs: $n \leq m+1$, so in practice, the time complexity of the algorithm by Schmidt approximates $O(m)$.

5.2 Merging multiple HV/MV transformers

Hitherto we limited ourselves to parts of the MV network having only one HV/MV transformer as supply point and in addition containing all nodes and edges connected to the HV/MV transformer (via a path). Then we can guarantee that this part of the network, called a *substation area*, is connected. The configuration of that part of the network is connected too, making it a spanning tree. In this way, we have that the substation area has the right properties to use the observations as presented in chapter 3 and in the previous subsection. However, it adds some extra information if we observe multiple substation areas at once. We first explain how and give the solution to this wanted generalization afterwards.

The relevance of considering multiple substation areas at once

Although the given approach for one substation area is an appropriate starting point, we would like to consider multiple substation areas at once, as this gives us additional information and so generates better solutions. This is because there are additional (open) cables between two different substation areas (section 1.2 described the layout of the MV network in general). Whilst substation areas cannot be connected to each other to avoid dangerous voltage situations, we could connect a substation area to a part of another substation area, if we decouple this part from the rest of that substation area. For this purpose, we can use the optional cables between the substation areas.

The existence of these optional cables and the possibility to change the 'classification' of the substation areas truly increases the reliability of the network, as we have more options to solve a problem than purely within the area where the problem arises. For example, if an edge e breaks down in substation area A and there is no appropriate switchover within area A (maybe there is no switchover at all or only a switchover giving unsuitable voltages or currents), there may be an appropriate solution using another substation area B (maybe there is sufficient remaining capacity in area B to feed some nodes in area A). Thus, the potential use of these optional cables between substation areas could really increase the reliability. If we consider these cables in our final algorithm, we ultimately give better insights in the reliability, as our results are more informative.

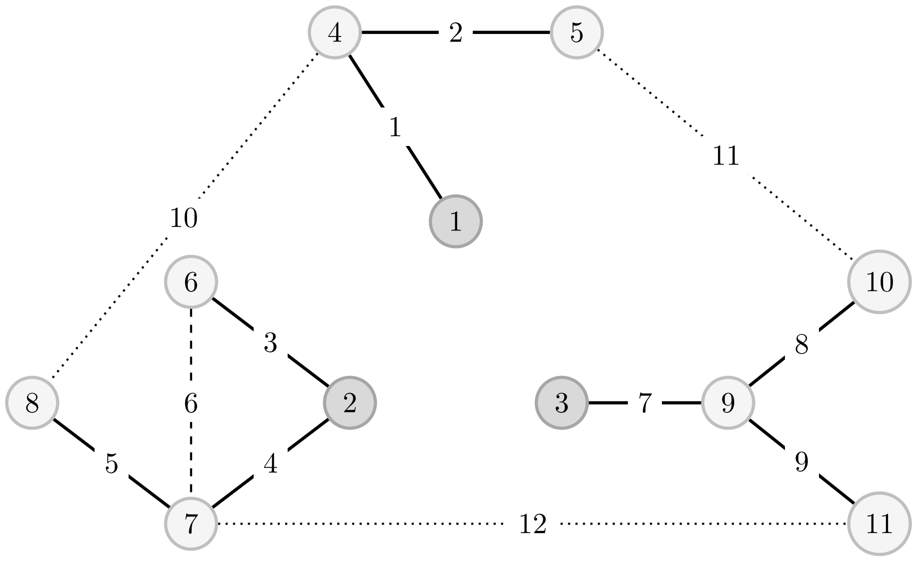
The solution: merging HV/MV transformers
Fortunately, finding the solution to combine multiple substation areas is not that hard. As Andrei and Chicco (2008) [6] describe, we can simply merge all supply nodes into a single node in the reduced network. In other words, we define all HV/MV transformers to be one and the same HV/MV transformer. Then the entire MV network is just one connected substation area and we can use all results as given before on the entire network at once.

Only the load flow computation still needs to use the original network, as different HV/MV transformers could have different voltage levels that cannot be combined into one value of the artificial 'super' HV/MV transformer. To ensure the correct computation of the load flow, we transform a solution of the new network to a solution of the original network. As we do not delete edges in the new network (only merge the HV/MV transformer node to just one node), we can use the same edge numbers in both networks to identify the edges back and forth in both networks. In this way, we can use the graph algorithms that use spanning trees on the new network, but calculate the load flow over the unique corresponding original network.

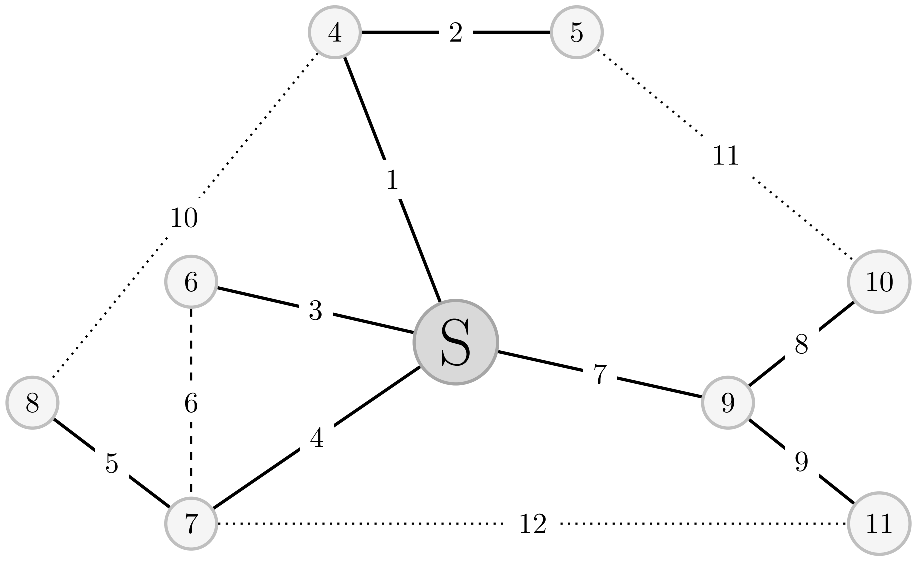
Below we give an example to make the merger comprehensible. Afterwards, we clarify the process in mathematical terms. Finally, we prove the correctness of this aggregation.

Example 5.6. We display a small MV network having three HV/MV transformers with node numbers 1, 2, 3. The MV network consequently falls apart in substation area (1) with nodes 1, 4, 5, substation area (2) with nodes 2, 6, 7, 8 and substation area (3) with nodes 3, 9, 10, 11. There are connections between the different substation areas, namely the optional edges 10, 11 and 12 (indicated with a dotted line). Edge 6 is an open edge too, but lies entirely in substation area (2).
The bold edges together form the actual configuration of the network. Notice that in each substation area the corresponding part of the configuration is a spanning tree. Further note that each load bus (a node not being an HV/MV tranforms, *middenspanningsruimte*) is connected via a path to precisely one HV/MV transformer.
Lastly, we perceive that the entire MV network (including the open edges) is connected.

This is not always true in general, for example the network without edges 10 and 11 is another possible MV network, in which we have two unconnected components.



We see that the existence of the open edges between different substation areas really increases the reliability of the network. For example, if edge 2 breaks down, we can feed node 5 if we turn on edge 11. Another example: if edge 7 breaks down, we could either switch edge 11 or edge 12 to provide each node with current. Of course, we should first check the results of the load flow calculation to see whether these switches are indeed acceptable.
In the following graph, we display the aggregation of the three HV/MV transformers. The node 'S' is the merger of the HV/MV transformers 1, 2 and 3 in the previous graph.



First note that no edge is removed or added with respect to the previous graph, so we do not need to rename the edges. Only edges ending in an HV/MV transformer now have a common end 'S', the artificial 'super' HV/MV transformer. If we save the original and new ends of the edges, we can easily switch between the two representations in a

bijective manner.
The bold edges still represent the actual configuration of the network (although the new graph is partially artificial). Note that the configuration is a spanning tree in this new graph.
Check yourself that the given switchovers using the original graph still suffice in this new graph.

Mathematical remarks
We first observe the new graph from a mathematical viewpoint. We prove the correctness of the merger afterwards.

By the merger, there cannot arise loops, as the original graph has no loops and there may not exist edges connecting two different HV/MV transformers (such an edge would form a loop in the new graph). The merging could create additional parallel edges, although it cannot create new parallel edges that are both in use. As the latter means there exist a vertex v directly connected to two different HV/MV transformers. As there may not exist a path between two HV/MV transformers, at least one edge connecting v and an HV/MV transformer should be open. Hereby the *configuration* in the new graph cannot have parallel edges, if we assume the configuration in the original graph has none.

Suppose a given MV network has h HV/MV transformers. Then the network consists of h substation areas. The actual configuration consists of the h spanning trees coming from the substation areas. Suppose the entire network has n nodes and substation area i has n_i nodes. Then the number of edges in the actual configuration equals $\sum_{i=1}^{h}(n_i - 1) = \sum_{i=1}^{h} n_i - h = n - h$. The number of edges in the configuration in the new graph is still $n - h$, although the number of nodes in the new graph equals only $n - h + 1$, as h HV/MV transformers are replaced by one substation S.

Note that the graph-theoretical name for the actual configuration in the original graph is a spanning *forest*, meaning a graph whose components are trees. (The classical definition of a forest as in Bondy and Murty (2008) [1], section 4.1, is just *an acyclic graph.*) A forest in the MV network has the additional property that each component contains precisely one HV/MV transformer. Now we can express the correctness of the merger of the HV/MV transformers in the following lemma:

Lemma 5.7. *Let G be an undirected graph representing an MV network having h HV/MV transformers. Let the actual configuration of G be the spanning subgraph C. Let G' be the graph created from G by merging all HV/MV transformers into one node called* S. *In addition, let T be the graph of the edges of C in G'. Then C is a spanning forest such that each component contains precisely one HV/MV transformer* iff *T is a spanning tree of G'.*

Proof. Suppose C is a spanning forest such that each component contains precisely one HV/MV transformer. Then C contains $n - h$ edges as argued before and T contains $n-h$ edges too. G' contains $n-h+1$ vertices in total. As each component of C contains precisely one HV/MV transformer and in G' all HV/MV transformers are merged into S, all nodes in T are connected (via a path) to S. Thus, T is connected. By proposition 2.13 (a) and (b), T is a spanning tree of G'.

Now assume T is a spanning tree of G'. Then T contains $n-h$ edges as G' has $n-h+1$ vertices. Hence C contains $n-h$ edges too. When S is split into the h HV/MV transformers (from T to C), we could destroy cycles in G'. However, this process cannot create new cycles in G that did not exist in G', because we only break connections between points and add no new ones. Therefore, as T does not contain a cycle, C does not either. By definition, C is a spanning forest and further C consists of h components as it has $n-h$ edges (exercise 4.1.4 in Bondy and Murty (2008) [1]). T connects each node to the substation node S, so in C each node is connected to at least one HV/MV transformer. As there are h HV/MV transformers in C and h components in C too, each component should contain precisely one HV/MV transformer to connect each node to an HV/MV transformer. $\square$

The final conclusion: if we find all spanning trees of the new graph, we know precisely all permissible configurations of the MV network (by transforming the spanning trees back to special forests in the original graph).

5.3 Adjusting the initial configuration to make it a spanning tree

This last section anticipates a problem that we will encounter in the next chapter, chapter 6, *Implementation in R & Practice*. The problem is as follows: although the initial configuration of the MV network should be a spanning tree after the merger of the HV/MV transformers, as described in the previous section, it is not a spanning tree in practice. The same holds for some part of the MV network consisting of a number of substation areas. Unfortunately, in several cases, the initial configuration is (1) not connected, (2) contains cycles, or both. To be able to use the algorithms and mathematical ideas as described in chapter 2, chapter 3 and this chapter, we need the initial configuration to be a spanning tree. In theory, it should be too and therefore we only search for spanning trees as reconfigurations of the network.

In this section, we describe how we could solve issues (1) and (2). We treat the mathematical side of the issues here, but treat the more practical side in the beginning of the next chapter. Note that from now on we mean by 'initial configuration' the configuration of the network *in the new graph* originating from the merger of the HV/MV transformers.

Solving problem (1): Find the separate components

If (1) is the case, the initial configuration contains unconnected parts, called components. In practice, the configuration consists of a few small components and a much larger component. Of course, it is very troubling if large parts of the network are disconnected from any HV/MV transformer and receive no electricity as a result. Therefore, this is generally not the case. Consequently, as all but one component is small, it is acceptable to remove these small parts from the graph to ensure the initial configuration is connected.

To be more specific, we completely remove all nodes and edges in these small components from the graph representing the MV network. Then these parts are deleted from the initial configuration too. The program R has a built-in function called *clusters* that returns all components of a graph given by their vertices. This function is in the package *igraph*. We do not discuss the implementation details of this function, but now we know problem (1) is relatively easy to solve using some built-in function of R.

Solving problem (2): Make the connected graph acyclic

If problem (2) holds, we need to delete some edges from cycles to make the initial configuration acyclic. However, we may not disconnect the initial configuration upon deletion of these edges, something what could occur. This requires some care, for which we need to use a special set of cycles of the initial configuration, for example *fundamental cycles*. Then we need to choose a suitable edge in each cycle to delete from the configuration. Such a method would work, if we have an efficient procedure to find a so-called *cycle basis*, for example of fundamental cycles, and delete edges from a *semi-ear decomposition* computed from the cycle basis. Of course, this is not immediately clear without giving the necessary definitions, but we could prove that this method works, using the theorems and proofs in the master thesis by Michiel van der Meulen (2015) [20]. In particular, we initially implemented this method in R and it gave indeed correct results. However, it turned out to have a slow performance. We explain why we could not conveniently speed up this implementation using a *cycle basis*.

If we want to make use of a built-in function in R to find a cycle basis, the only suitable function is *fundCycles* in the package *ggm* (an extension of the package *igraph*). This function has a slow performance: it takes almost 40 minutes to run on the entire MV network of about 60.000 cables. The only other option to use cycle bases is to implement a function to find a cycle basis ourselves. It should be efficient as it has to function on the entire network. We could do this, but it could be quite time consuming to make the function fast. Moreover, it is not the topic of this thesis. It is only necessary because the assumption of the initial configuration is not entirely correct and we want to know the results of our final algorithm on the MV network. Luckily, there is another method to turn the graph in an acyclic connected graph, i.e. to make sure that the initial configuration is a spanning tree. This method employs a breadth-first search on the graph.

We could perform a breadth-first search, as the graph is already made connected after solving problem (1). We begin this breadth-first search in the artificial substation node S (as defined in the previous section). First of all, such a tree search will always end in a spanning tree as the graph is connected, as we mentioned in section 2.2. This breadth-first search is sufficient for our purposes, as we need the initial configuration to be a spanning tree, but we do not really need to know the cycle basis. Finding the cycle basis was a means, not an end. Furthermore, a tree search is very efficient, as we need to visit every node only once. As proved in section 5.1, the time complexity of such a search is linear, i.e. it takes $O(n + m)$ time.

The reason why we perform a *breadth*-first search instead of a *depth*-first search is that the initial configuration of an MV network resembles a BFS-tree in all likelihood. The electricity distribution network seems to be more reliable if the current is distributed into multiple branches from the source (an HV/MV transformer). If the current had to go through a long piece of consecutive cables before it is further distributed, the network would be more susceptible to major outage if a cable near the source breaks down. It seems preferable to better distribute the cables and hence the current. This idea corresponds to a breadth-first distribution.

All in all, if we conduct a breadth-first search on the connected graph, we make the connected graph acyclic and it complies with the assumption. Furthermore, this procedure is fast and it seems to match the shape of an electricity distribution network.

Chapter 6

Implementation & Overview

This chapter treats with the implementation of a solution to the $m-1$ problem in the program R. It also presents the link between the previous chapters, in particular how the given algorithms and reductions are interrelated in the eventual algorithm.

First, we present the preparation of the MV network and the initial configuration, which is required before running the actual algorithm to check the $m-1$ principle, hereinafter referred to as '$m-1$ algorithm'. Second, we reflect on the combination of the reductions (k and Andrei-Chicco) and algorithms (Gabow and Myers, Schmidt). Thereafter we present an overview of the final algorithm and explain the steps therein. The last section addresses some other practical matters regarding the implementation

We try to explain the design of the code without explicitly presenting the code. In case you are interested in the actual code, please contact the author. This chapter provides the overall idea of the implementation, but avoids focus on the implementation details. On the other hand, we present some facts about the implementation in R in the last section.

Two notational remarks: instead of HV/MV transformer, we often write in the code 'OS', the abbreviation of the Dutch name 'onderstation'. Both HV/MV transformer and OS refer to a slack bus of a part of the MV network, or equivalently, a connection of the MV network with the high voltage network.

Similarly, by 'MSR', the abbreviation of the Dutch name 'middenspanningsruimte', we refer to a load bus in the MV network. For convenience, we call every node that is not an OS an MSR (also the so-called sockets, in Dutch 'moffen', that are strictly speaking not an MSR). Mostly, an MSR is a connection between the MV network and the low voltage network. In fact, an MSR is a slack bus of the low voltage network. However, this last note is not within the scope of this project.

6.1 Preparing the MV network and initial configuration

As noted before, although the initial configuration of the network should be a spanning tree (after the merger of the HV/MV transformers as described in section 5.2), this is generally not the case. After the merger of the HV/MV transformers, the initial configuration might (1) not be connected, (2) contain cycles, or both. Section 5.3 described how to solve these problems mathematically. This section presents all preparation steps in the implementation in R. The R script 'Select MS, prepare data' performs this preparation. First, we note the required conditions to the input data. Second, we display an overview of the preparation of an MV network and we clarify each step afterwards.

6.1.1 Required conditions to the input data

We need the following information anyhow to be able to extract the relevant information in the $m-1$ algorithm: a list of unique edges (the cables) in the MV network of which we know the ends, i.e. the nodes. The edges and nodes should have unique names to be able to distinguish them and avoid computation of ambiguous results. For the data of Alliander, we take the so-called 'rail keys' of the nodes as unique node names. The enclosure number ('behuizingsnummer') of the nodes, which is used for the link with other information tables, does not guarantee this uniqueness property. We identify the edges by giving them a unique number.

Furthermore, we need to know which cables are in use (closed) or not (open). Then we can derive the configuration of the network. Graph-theoretically, the above provides sufficient information already.

On the other hand, we need to check the physical conditions of a reconfiguration, through the computation of the load flow. Hence we need many different physical values. For the cables, we need to know the impedance and (absolute maximum) current capacity. For the OS's (a part of the nodes), we need the voltage value. For the MSRs (the remaining nodes), we need to know the values of the power consumption, the minimum voltage capacity and the maximum voltage capacity.

To cut the MV network into smaller, appropriate areas, we need to know the names of the substation areas (in Dutch 'OS-gebieden'). In particular, we need to know to which substation area each edge belongs. Then we can select all edges from certain substation areas if we want to. In practice, we either select Alliander's MV network completely or we select the MV cables of any number (and combination) of substation areas. Considering smaller parts than substation areas is not that interesting for this research (in that case the MV network is too small). For Alliander's MV network, the 'route name' of each edge provides the substation area name the edge belongs to. Some rows of an example table comprising the required information to prepare the data:

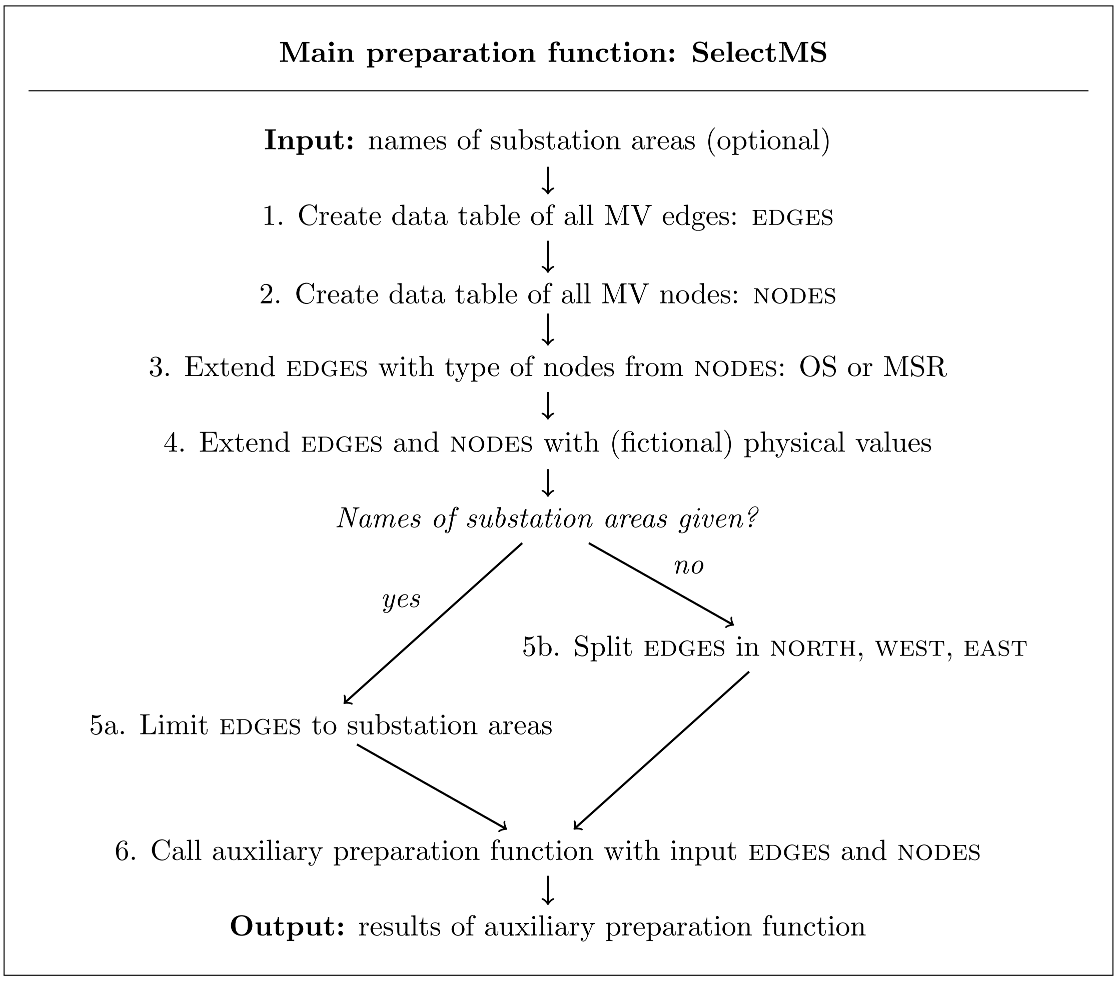
Edge	**From rail key**	**To rail key**	**Grid opening**	**Route name**
1	6 002 809_10-1RA1	6 005 364_10-1RA1	closed	AMO 10-1V2.14
2	6 018 134_10-1RA1	6 005 364_10-1RA1	open	AMO 10-1V2.14
3	6 003 030_10-2RA2	6 005 234_10-1RA1	closed	AMO 10-1V2.02
4	6 008 656_10-2RA2	9 006 972_10-1RA1	closed	KUN 10-1V11

Note that this table does not present the physical conditions of the initial network.

Lastly, we need to know the type of node each rail key corresponds to: OS or MSR. This information is also not yet present in this example table. In the case of Alliander, we use the data table of all MV edges as present by IntellEvent.

6.1.2 Overview of the preparation functions

We present an overview of the main preparation function. Afterwards, we elaborate on some steps in the function. The main function uses an auxiliary function, which we illustrate thereafter.


Main preparation function: SelectMS

Input: names of substation areas (optional)

1. Create data table of all MV edges: EDGES

2. Create data table of all MV nodes: NODES

3. Extend EDGES with type of nodes from NODES: OS or MSR

4. Extend EDGES and NODES with (fictional) physical values

Names of substation areas given?

yes

5a. Limit EDGES to substation areas

no

5b. Split EDGES in NORTH, WEST, EAST

6. Call auxiliary preparation function with input EDGES and NODES

Output: results of auxiliary preparation function

Explanatory notes to the overview of the main preparation function:

- The data table EDGES is similar to the example table given above. It consists of at least the columns in the example table. The data table NODES consists of the node names (in Alliander's case: the enclosure numbers, although the rail keys would be better) and the function of the nodes: OS or MSR. As already mentioned, EDGES results from the project IntellEvent. Furthermore, NODES comes from 'BAR_GIS'.
- We could consider EDGES and NODES as input data instead of data that is formed in the main preparation function (in steps 1 and 2). We could change the function

to that shape. The reason that we create the data tables only in the function is that the data tables can be automatically loaded from a database and then consist of the most recent state of the MV network. The only input variable of the function is therefore a string of substation area names (sometimes simply called OS names). This variable is optional, so we could omit it too.

- Step 3 extends EDGES with the type of nodes from NODES in the sense that we add the type of node to each end of each edge. More specifically, we append EDGES with two columns 'From group' and 'To group' to display the type of the first and second end of each edge, respectively.
- Unfortunately, the values of most for the physical quantities are not yet present in the data table by IntellEvent. Thus, we must come up with the values ourselves. Of course, we try to do this realistically, but this should be improved if we want to use the algorithms in practice (although this does not effect the correctness of the algorithms).
- Step 5a easily selects all edges lying in specific substation areas using the given OS names and the route names in EDGES. Step 5b splits EDGES in three parts based on three strings of OS names, each representing all substation areas in each of the three regions of Alliander's MV network. NORTH, WEST and EAST are mutually disjoint and together cover the whole MV network.

  We need to split the whole network in smaller parts because the network (consisting of about 60.000 edges) is too large to work with in R. For example, we need to use an $n \times n$ adjacency matrix as input in a breadth-first search to prepare the spanning tree in the auxiliary preparation function (step 7 below), but matrices having dimensions larger than about 25.000×25.000 cause problems in R. Note that $n < m$ but they differ only several thousand.

  We split the network in these three parts (NORTH, WEST and EAST) because these three parts are mutually unconnected. The parts are also quite evenly distributed in terms of size and they are clearly distinguishable geographically. As the three parts are mutually unconnected, the results of the parts together are equal to the results of the MV network in its totality. Finally, note that we separately perform step 6 for all three parts with associated EDGES.

The number of nodes n, the number of edges m and the number of optional edges r in the three geographical parts of the whole MV network are:

Region	n	m	r
NORTH	9300	10200	900
WEST	22500	25300	2800
EAST	21400	24000	2600
Total	53200	59500	6300

Note that the precise numbers change over time: by work on the MV network some pieces of the network may be disconnected and temporarily removed from the network. However, this is hardly ever the case for larger parts of the MV network.

On the next page we display the auxiliary preparation function, which is used in the main preparation function. Below, we present the (first) explanatory notes to the overview of the auxiliary preparation function:

- Step 1 removes edges from the high voltage network that are incorrectly present in the MV network.
- Step 2 merges all OS's into one OS called 'OS', as described in section 5.2.
- Step 3 removes open edges *having an end of degree one* from the network, as these edges disconnect the configuration of the network. In most cases, such edges exist because we limit the MV network to some part consisting of a number of substation areas. If a neighbouring substation area is not selected in this network, then some of the open cables in the network are connected to this substation area. However, as we did not select the substation area, these open cables are connected to a single node and disconnect the graph. We call such edges open edges at the ends of the network.
- Similar to the opening of an edge of each cycle in what will become the spanning tree, we open parallel edges (in step 4) to ensure the configuration of the network does not contain parallel edges, which are a special kind of cycles. Of course, we keep one of the parallel edges closed if there was at least one parallel edge in use.
- Step 5 derives three data tables and an array from EDGES. The $m-1$ algorithm needs these inputs, except for EDGES_RED. We briefly explain the usefulness of each part. For the graph-theoretical part of the $m-1$ algorithm where we search for spanning trees and bridges, among others, we need an abstraction of the MV network in which we represent the edges and nodes as consecutive numbers. For this purpose we defined TRANSEDGES, the abbreviation of 'translated edges'. We save EDGES_RED because it directly corresponds to TRANSEDGES, only having the original node names. Where EDGES_RED has one merged OS, EDGES_ORG still has all original node names. EDGES_ORG is not only useful to know the original nodes, but also includes the physical values of the edges needed in the load flow computation. TREE is just an array consisting of all edges in the initial configuration (that will be a spanning tree after step 7). Although EDGES_RED and EDGES_ORG actually also contain the column 'Grid opening', TREE is an easy way to represent the configuration (spanning tree) of the network. It can easily be adapted to another spanning tree (in the $m-1$ algorithm) too.
- We already explained the need for steps 6 and 7 in section 5.3. We presented the mathematical solutions to the problems there too.

Auxiliary preparation function: PrepareEdgesTreeEtc

Input: EDGES, NODES

$\downarrow$

1. Remove edges that connect two OSs from EDGES

$\downarrow$

2. Merge all OSs into one OS in EDGES

$\downarrow$

3. Remove open edges at the ends of the network from EDGES

$\downarrow$

4. Open parallel edges in EDGES

$\downarrow$

5. Derive four useful parts from EDGES (first four in table below)

$\downarrow$

6. Delete the small unconnected components in TREE from all parts

$\downarrow$

7. Open an edge from each cycle in TREE

$\downarrow$

8. Make a table of the network's MSRs, including physical values: MSRLIST

$\downarrow$

9. Make a table of the network's OSs, including physical values: OSLIST

$\downarrow$

10. Make a table of the network's MV links between substation areas: OSGRAPH

$\downarrow$

Output: list of seven parts:

Part's name	**Features**
EDGES_RED	table of edges with their merged ends (MSRs given by their so-called railkeys, OSs all given by same name 'OS')
EDGES_ORG	table of edges with their original ends (given by railkeys), including values of physical quantities 'Impedance' and 'Current Capacity', and including the route names
TRANSEDGES	table of edges having simplified node names: numbers $1, \ldots, n$, TRANSEDGES corresponds to EDGES_RED
TREE	array of edge numbers of edges in initial spanning tree
MSRLIST	table of the network's MSRs (given by railkeys), including values of physical quantities 'Power Consumption', 'Minimum Voltage Capacity' and 'Maximum Voltage Capacity'
OSLIST	table of the network's OSs (given by railkeys), including physical 'Voltage Value'
OSGRAPH	table of the network's MV links between substation areas

- In the linear load flow computation, part of the $m-1$ algorithm, we need lists of MSRs and OSs including associated physical values. That is why we execute steps 8 and 9.
- As mentioned in subsection 3.3.1, we want to determine only the useful k reductions to limit the number of k reductions in the $m-1$ algorithm. These useful k reductions arise from the combinations of $\lceil \frac{k}{2} \rceil$ optional edges within substation areas and the combinations within neighbouring substation areas. To be able to compute these useful combinations, we need a table of links between substation areas. This is OSGRAPH, which we make in advance in this preparation function. In particular, OSGRAPH consists of three columns: two columns to identify the substation names (OS names) and a column consisting of the edge number of the MV edge that connects the areas.

A last important note concerning the different parts (results of the preparation functions) is the uniformity of the edge numbers: the edge numbers in EDGES_RED, EDGES_ORG and TRANSEDGES all concern the same edges. Only the depiction of the nodes is different. Likewise, the edge numbers in TREE and OSGRAPH correspond to the same edges. In this way, we can easily swap between different representations of the MV network (the original network or the mathematical network where we merged the OSs). In particular, the edges are always identified by the unique edge numbers $1, \ldots, m$.

The remainder of this chapter focuses on the actual implementation of the $m-1$ algorithm, i.e. it assumes that we know the required data and it treats the actual computations to check the $m-1$ principle. Keep in mind that the input data of the $m-1$ algorithm is the output data of the main preparation function given above.

6.2 Combination of reductions and algorithms

Before we present the overview of the $m-1$ algorithm in the next section, we reflect on the combination of the reductions and mathematical algorithms that are part of the total algorithm. Although it is probably clear that we need the reductions (k and Andrei-Chicco) and the algorithms (to find all spanning trees, to find all bridges), it is not immediately clear how we combine these parts. This section argues why we came to order these mathematical tools in the final algorithm in the specific way we did.

First, we reason about the order of the spanning tree listing, k reductions and bridges. After an intermezzo about the Andrei-Chicco reduction applied to Alliander's MV network, we continue the reasoning on the order of the mathematical tools by adding the Andrei-Chicco reduction. Last, we mention some consequences of the given order and explain how to deal with those.

6.2.1 Combination of spanning tree listing, k reductions and bridges

As we proved in section 3.2, the number of spanning trees grows very rapidly with the number of edges for a fixed number of nodes. As the table of Alliander's MV regions in the previous subsection (subsection 6.1.2) depicts, the number of nodes and edges in Alliander's MV networks is not suitable for the use of a spanning tree enumeration algorithm (in our case Gabow and Myers' algorithm) at once. If we try to compute the number of spanning trees of the northern region of Alliander's MV network for example, using our implementation in R of Kirchhoff's theorem (theorem 3.8), then the implemented function returns 'Inf'. This means that the number of spanning trees is too large a number for R. Thus, before we can apply Gabow and Myers' algorithm, the network must be reduced significantly.

We begin with zooming in on the use of the k reduction procedure. As explained in subsection 3.3.1, the k reductions ensure that we enumerate only suitable spanning trees of the MV network, instead of all spanning trees. As the number of spanning trees grows root exponentially with $2m - 2n$, the k reduction method is very appropriate if it reduces the difference $m - n$ significantly. This is the case for a small k and a large initial $m - n$, because the number of edges reduces to $n - 1 + \lceil \frac{k}{2} \rceil$ in a k reduction (and n does not change). $k \leq 6$ in Alliander's case and the initial $m - n$ is in the range of thousands, as can be seen in the table in the previous subsection. As a consequence, the k reduction procedure is very appropriate for Alliander's MV network. The disadvantage of the k reduction method is that it requires many iterations in which $\lceil \frac{k}{2} \rceil$ optional edges are added to the initial spanning tree. We need all these iterations to be sure we have examined all useful spanning trees and hence all possible switchover combinations. Fortunately, we can manage this number of k reductions by using the OSGRAPH, so examining only the informative combinations of optional edges. These are the combinations within neighbouring substation areas.

Let us take a look at Schmidt's bridge enumeration algorithm. The use of this algorithm is to determine the graph-theoretically unswitchable edges beforehand. This has two advantages. First of all, we can distinguish graph-theoretically unswitchable edges from edges that are unswitchable because of the load flow (the exceeding of a voltage or current capacity). Second, if there are no edges that do not have a switchover because of the load flow, then the $m - 1$ algorithm terminates before the execution of all k reductions and we save computation time. We do not need to search for non-existing switchovers in the k reductions, for we computed the bridges (graph-theoretically unswitchable edges) beforehand. As follows from the above, we should perform Schmidt's bridge enumeration algorithm before the k reduction procedure (and thus before the spanning tree listing), for then it could save computation time in the k reductions.

The combination of the k reduction procedure and two algorithms is a good start to solve the $m - 1$ problem, but it is still insufficient to solve the problem for large areas of Alliander's MV network. The computations in each k reduction still take too much time. We cannot obtain more information or save computation time by deploying the spanning tree listing, k reduction procedure or bridge enumeration more than once (except that we do use the spanning tree enumeration algorithm in each k reduction, but we took this into account already). Another reduction is indispensable and we will use the Andrei-Chicco reduction. Adding this reduction is more complicated, for it could be deployed at different places in the $m - 1$ algorithm and could be used more than once. We first assess the Andrei-Chicco reduction applied to the MV networks in more detail. Hereafter we discuss at which stages to add the Andrei-Chicco reduction in the final algorithm.

6.2.2 Considering the Andrei-Chicco reduction for Alliander's MV network

As mentioned before, the number of edges and nodes are still too large if we only use the k reduction. Using the Andrei-Chicco reduction, we could significantly reduce these numbers. We present the results of the Andrei-Chicco reduction applied to the three large regions of Alliander's MV network below. Note that we abbreviate the Andrei-Chicco reduction by 'ACR'. We present the reduction factors of n and m too, abbreviated by rf(n) and rf(m), respectively.

Region	n	m	n_{ACR}	m_{ACR}	rf(n)	rf(m)	comp. time
NORTH	9300	10200	800	1500	11,6	6,8	3 sec
WEST	22500	25300	3000	5600	7,5	4,5	13 sec
EAST	21400	24000	3400	5900	6,3	4,1	11 sec
Total	53200	59500	7200	13000	7,4	4,6	27 sec

There are two reasons why the Andrei-Chicco reduction works well for Alliander's MV network. First, there are a lot of edges in branches (in Dutch 'uitlopers') in the MV network. We cannot switch these edges, so we can immediately save this outcome after the first step of the ACR. More importantly, we can delete these edges in branches from the graph as they do not influence switchovers for other edges. The edges in branches are in every spanning tree and due to there place in the ends of the graph, they do not disconnect the graph upon removal.

Second, there are a lot of nodes having degree two in Alliander's MV network. Those nodes are located in paths (as described in section 3.3.2) that we will contract to single edges or loops in the Andrei-Chicco reduction. As as result, we remove the inner nodes and edges of these paths and the ACR graph is much smaller than the original graph.

As mentioned in section 3.3.2, in general the Andrei-Chicco reduction does not reduce each graph a lot. The large reduction is dependent on the number of edges in branches and the number of nodes of degree two. For electricity distribution networks, these two numbers are generally high. As shown in the table above, although the reduction factors differ considerably between the MV regions, it is substantial in general.

The differences between the reduction factors of n and m (in the table) are striking. rf(n) is considerably larger than rf(m). This is because the number of residual edges $r = m - n + 1$ does not change that much in the ACR. Henceforth the ratio $\frac{m}{n} = \frac{n+(m-n)}{n} = 1 + \frac{m-n}{n}$ grows after the ACR, because $m - n$ remains more or less constant, while n decreases. Therefore, n becomes relatively smaller than m. Consequently, the ACR reduced n more than m.

We will explain the two causes of the little change of the number of optional edges. First of all, there exist no optional edges in branches, as these disconnect the initial configuration of the graph. Besides that, there is at most one optional edge in a path of edges that reduces to one edge in the ACR, for the middle part of the path would be disconnected otherwise. However, by reducing this path to one edge, only the optional edge remains to maintain a comparable graph structure. The only optional edges that we will remove in the Andrei-Chicco reduction are the optional edges in formed loops, as we remove all these loops completely. However, there are relatively few loops among the reduced paths.

Now that we have looked at the Andrei-Chicco reduction in our case studies, we consider the stages where we could add it in the $m - 1$ algorithm.

6.2.3 Adding the Andrei-Chicco reduction

We display an overview of the final combination of the reductions and algorithms in the $m - 1$ algorithm. Afterwards, we argue why we added the Andrei-Chicco reduction at the indicated stages.

Combination of reductions and algorithms in $m - 1$ algorithm

$$\begin{tikzcd}
\text{Andrei-Chicco reduction of original graph} \arrow[d] \\
\text{Schmidt's bridge enumeration algorithm} \arrow[d] \\
k \text{ reduction} \arrow[d] \\
\text{Andrei-Chicco reduction of } k \text{ reduced graph} \arrow[d] \\
\text{Gabow and Myers' spanning tree enumeration algorithm} \arrow[uu, bend right=60]
\end{tikzcd}$$

We could Andrei-Chicco reduce the graph either before we perform a k reduction or after the k reduction. We present the benefits of both. An advantage of an Andrei-Chicco reduction before the k reductions is that we only have to reduce the graph once instead of in each iteration. Although the computation time of the ACR is not that much (a few seconds as can be seen in the table on page 95), repeating the reduction in each k reduction will eventually cost a lot of time. Therefore, we need to perform the Andrei-Chicco reduction at least once before the k reduction procedure.

In practice, the computation time of Schmidt's bridge algorithm is much shorter if we compute the bridges of the Andrei-Chicco reduced graph. However, we need to transform the bridges of the reduced graph to bridges of the original graph. Fortunately, there is an easy rule for that: if an edge e is a bridge in the ACR graph, then all edges in the path that was reduced to e are bridges of the original graph. Furthermore, if an edge e is a bridge in the original graph, then all edges in the path where e belongs to are bridges and the path edge in the ACR graph is a bridge too. Hence, the translation of bridges in the ACR graph to bridges in the original graph does not last long. Another benefit of performing the Andrei-Chicco reduction before Schmidt's algorithm is that we have determined and deleted all edges in branches beforehand. Doing so, we distinguish the edges in branches from other kinds of bridges, which may be interesting. Schmidt's algorithm would detect the edges in branches too (if we would perform it before the ACR), as an edge in a branch is a special kind of bridge. However, it would not determine the difference between the kinds of bridges.

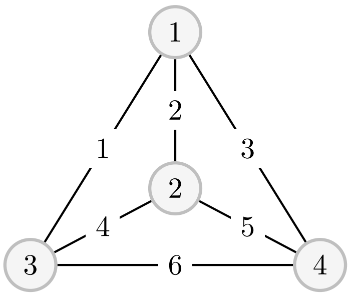
An advantage of an Andrei-Chicco reduction after the k reduction is that a lot of edges will disappear because they are in branches in the k reduced graph (important note: they are not necessarily in branches in the original graph). Hence we do not have to compute these edges in the spanning trees again and again, whilst they do not produce switchover information. The reason that there are so many edges in branches is that there are at most $\lceil \frac{k}{2} \rceil \leq 3$ optional edges in the k reduced graph. As a consequence, there are few cycles in the graph and thus a lot of bridges by lemma 5.3, of which, in practice, most of the bridges are edges in branches.

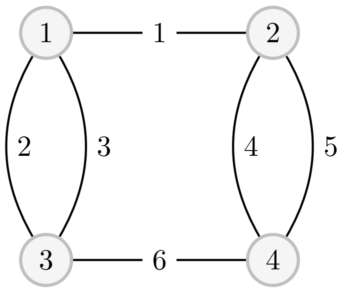
For the sake of clarity, once more note that we can only use an Andrei-Chicco reduction after each k reduction if we Andrei-Chicco reduce the graph before the k reduction too. For the computation time of the Andrei-Chicco reductions in the k reduction part of the algorithm must be short enough. It turns out that if we Andrei-Chicco reduce the graph before the k reduction too, then the computation time of an ACR in a k reduction takes less than a second.

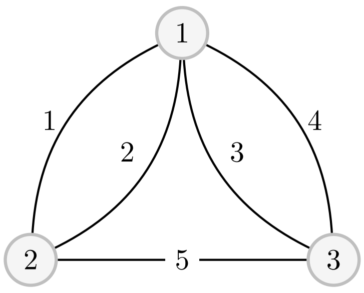
Based on practical considerations, namely the computation time, we embedded an Andrei-Chicco reduction both before and after the k reductions. Last, we explain why we did not embed more Andrei-Chicco reductions in the $m-1$ algorithm.

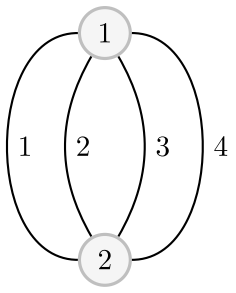
Another Andrei-Chicco reduction before the k reductions is quite useless because it barely reduces the MV network further. This reduction is only effective if there were a lot of loops formed in the first ACR, but this is not the case in practice. Thus, it is a waste of time to do this additional Andrei-Chicco reduction.

Furthermore, another Andrei-Chicco reduction after each k reduction has little use, because the present (k reduced and single Andrei-Chicco reduced) graph is already very small. In the extreme cases where we have three optional edges in the reduced graph and we did not form loops in the Andrei-Chicco reduction (we would delete those beforehand), the k reduced and single Andrei-Chicco reduced graph has one of the following shapes:









As no loops were formed in the ACR, the cycles in the graph have to be interconnected, i.e. each cycle should intersect another cycle with at least one edge. As each cycle in the graph should contain an optional edge (because the graph without the optional edges is the initial spanning tree), a so-called cycle basis of the graph consists of only three cycles. In particular, the cycles in the cycle basis intersect with at least one other cycle in the basis. Working out the different ways in which such three cycles could relate to each other, we find the four possible graphs as k reduced and single Andrei-Chicco reduced graphs as a result. Note that we have removed all edges in branches even as all nodes of degree two in the first two steps of the ACR.

Using Kirchhoff's theorem (theorem 3.8) we find the number of spanning trees of the four graphs to equal 16, 12, 8 and 4, respectively. We see that another Andrei-Chicco reduction of such a k reduced and single Andrei-Chicco reduced graph is not necessary. In particular, we cannot even reduce the graphs further in these extreme cases, as there are only vertices of degree ≥ 3.

Last, note that some other graph shapes are possible. However, in those cases, less than three optional edges remain in the reduced graph. There could be a few edges in branches in these graphs that we actually could remove using an additional Andrei-Chicco reduction. However, the graph will never be larger than all four shapes given above. Consequently, they will also not contain more spanning trees.

All in all, we conclude that the combination of the reductions and algorithms given on page 96 is the most suitable for Alliander's MV network. The next section (section 6.3) treats the structure of the $m-1$ algorithm in more detail. It explains the intermediate steps needed to correctly build those parts together and we explain some other logical rules we use. Before the next section, we give some important consequences of the chosen combination of reductions and algorithms.

6.2.4 Consequences of the combination of reductions and algorithms

We mention three important mutual effects of the algorithms and reductions on each other. Two require some further explanation and will be dealt with in separate sections below. First of all, observe that the edge names get different meanings by Andrei-Chicco reducing the network. Therefore, we have to take care of the edge numbering. Furthermore, both the Andrei-Chicco reduction and Gabow and Myers' algorithm assume edge numbers $1, \ldots, m$ and node numbers $1, \ldots, n$. As the k reduction and Andrei-Chicco reduction could destroy this feature (initially this feature holds by the definition of TRANSEDGES), we have to rename the edges and nodes multiple times. We remember the original edge and node names, so we could switch between the different names.

Considerations on the combinations of switchovers from ACR structures
Besides the numbering, we have to interpret the results on an Andrei-Chicco reduced graph correctly as results on the original graph. We described how to do this in subsection 3.3.2. In particular, we have to combine the switchover information from the Andrei-Chicco reduction itself and from new spanning trees correctly, as described at the end of that subsection. This calls for a balance between computing all possible switchover combinations at least once, but not computing such switchover combinations again and again, whereby we increase the computation time a lot. We assure this by adjusting the first Andrei-Chicco reduction a bit: we prevent the formation of loops and consequently do not remove loops.

Due to the fact that we need to combine switchovers within loops with switchovers within paths or switchovers from spanning trees, we need to store all switchovers arising from loops and combine them with all other switchovers (as long as k is not exceeded) thereafter. Therefore, we artificially prevent the formation of loops by reducing a path starting and ending in the same node to two parallel edges (instead of a loop).

As a result, we only have to consider switches within paths (and not within loops) at the first Andrei-Chicco reduction. In addition, as the ACR does not remove paths, even though they are contracted to one edge, the paths are still saved via the categories if we perform another Andrei-Chicco reduction. Although we have to combine the categories of two reductions, we can still derive all switchover information within paths using the second Andrei-Chicco reduction (in combination with both categories). This saves a lot of work, for we only have to combine the structures of the second Andrei-Chicco reduction instead of those of the first reduction too. Thus, in the eventual algorithm we first perform a limited Andrei-Chicco reduction without loops and the original reduction including loops during the k reductions afterwards.

Note that one could still think that the computation of the switchover combinations occurring from loops, paths and spanning trees would take too much computation time. It seems like this recombination of the contracted structures from the reduction (loops and paths) is undoing the Andrei-Chicco reduction in a way. We acknowledge that this is a point of attention. The important observation is that we save all switchovers found from loops, paths and spanning trees after the second Andrei-Chicco reduction, but we try the combinations only for edges that do not have a switchover yet (or we determined that they have no switchover at all). For trying switchovers for already determined edges is a waste of time.

In particular, we store the switchovers in two lists: one list saves switchovers for edges that do not have a switchover yet (so these switchovers must be unacceptable with respect to the load flow), the other list saves switchovers for edges that already have a switchover. We only try the combinations of switchovers for undetermined edges, but possibly using switchovers of determined edges.

Besides, note that we only combine the switchovers found in one iteration of the k reduction procedure. As described in subsection 3.3.1, we make sure we only compute useful k reductions. We only use combinations of optional edges in the k reduction that lie in the neighbourhood of each other and hence really have an effect on each other. As the graph after the second Andrei-Chicco reduction is very small, as explained in the previous subsection (subsection 6.2.3), the computation time of the switchover combinations within one k reduction is not that much.

Concluding, the computation of combinations of switchovers (including the load flow checks) takes some time. However, to guarantee the completeness of the $m-1$ algorithm we have to do this. The method presented above is probably the best way with respect to the computation time. (At the very least, it is the best performing solution in our implementation of the algorithm.)

Case distinctions and the splitting into different values of k

By the previous observation, we have to combine all kinds of combinations of switchovers from loops, paths and spanning trees. As the number of combinations is bounded by k, the required combinations differ for different values of k.

For example, if $k = 3$, then we only need to combine two switchovers both using one switch. Suppose we can theoretically (not by the load flow) switch edge a by b and suppose we can switch c using d. Then we could possibly switch a using b, c and d. In that case, we use precisely three switches, which is the maximum number. However, if $k = 5$, then we could combine two or three such switchovers. Furthermore, we could also combine a switchover using three switches and a switchover using one switch. (Note that the only earlier found switchovers using three switches come from the spanning tree strategy. Switching within a loop or a path always uses one switch.)

Consequently, the case distinctions for combining switchovers differ per k. In particular, the number of case distinctions increases rapidly if k grows. Therefore, we split the eventual $m-1$ algorithm for different values of k.

If $k = 1$, we do not need any combination of switchovers. Besides that, the cases $k = 2p+1$ and $k = 2p+2$ for $p \in \mathbb{N}$ are equal, as noted before in section 2.3, as we cannot switch half an edge and we need one more edge to close than to open. The next section presents the different variations of the $m-1$ algorithm for different values of k.

As Alliander takes $k \leq 6$, we distinguish the cases $k = 1$, $k = 3$ and $k = 5$.

6.3 Overview of the final algorithm

We will present the different versions of the $m - 1$ algorithm, depending on k. We are only interested in $k \in \{1, 3, 5\}$. Although the algorithm is simpler in the case $k = 1$ than in the other cases, we first present the cases $k = 3$ and $k = 5$, as those algorithms closely resemble the general approach described in the previous section. The case $k = 1$ simplifies insofar that we do not even need new spanning trees. We also present the case where we do not compute the load flow, which is useful if we do not know the physical values or if we want a quick check which edges are theoretically switchable (including which edges to switch). The case without the load flow is similar to the case $k = 1$.

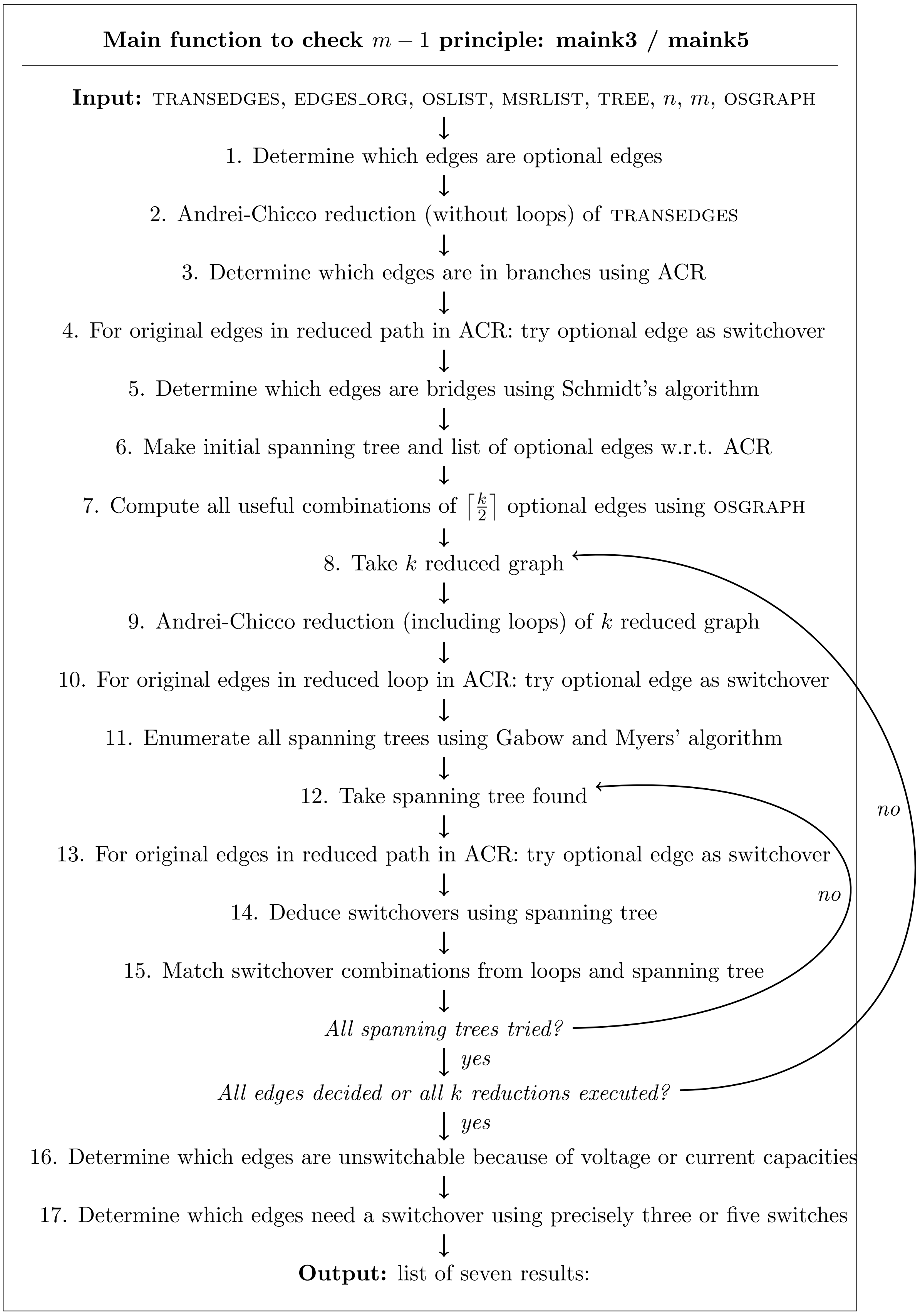
Before we display the overviews, we make an important remark. The goal of the $m - 1$ algorithm is to find out which edges are switchable. In particular, we want to know which edges we need to switch for the switchable edges and, for edges that are not switchable, we want to understand why it is not. Therefore, it is sufficient to stop searching for a switchover for a particular edge if we already have a suitable switchover for that edge. The $m - 1$ algorithm is not intended for *optimising* which switchover is the best one if there is at least one. (We first need to define the criteria of what is the 'best', of course.) One could adapt the $m - 1$ problem and algorithm to make it an optimisation problem. However, this would (drastically) increase the computation time, as we need to evaluate every possible switchover, instead of stopping if we have found a suitable one (for each edge). Concluding, optimising the switchovers is not the topic of this project. Notwithstanding, we optimise the switchovers in a sense: if we found a switchover using more than one switch, we keep on searching for other switchovers using less switches. Therefore, if we executed all k reductions (this is not always the case), then we know there is no switchover using less switches than the finally saved switch (for a particular edge). In this way we avoid unnecessarily cumbersome solutions. During the execution of the algorithm, we save the switchovers in the list OUTPUT, which is updated with shorter switchovers during the algorithm too.

The two subsequent subsections present an overview of the $m - 1$ algorithm for the given values of k. Afterwards, we clarify some of the steps.

6.3.1 The cases $k = 3$ and $k = 5$

We present an overview of the $m - 1$ algorithms for the cases $k = 3$ and $k = 5$. The actual implementation for the two cases differs slightly, mainly in the case distinctions for the switchover combinations (from different ACR structures as described in 3.3.2 and 6.2.4). However, the general structure of both variations of the algorithm is very similar. We depict the overview on the next page.

First note that we neither display the computation of the load flow, nor the check whether a voltage or current capacity is exceeded. We perform this computation every time we found a possible graph-theoretical switchover. If the voltage and current capacities are not exceeded, we save the switchover in the list OUTPUT.


Main function to check $m-1$ principle: maink3 / maink5
Input: TRANSEDGES, EDGES_ORG, OSLIST, MSRLIST, TREE, n, m, OSGRAPH
1. Determine which edges are optional edges
2. Andrei-Chicco reduction (without loops) of TRANSEDGES
3. Determine which edges are in branches using ACR
4. For original edges in reduced path in ACR: try optional edge as switchover
5. Determine which edges are bridges using Schmidt's algorithm
6. Make initial spanning tree and list of optional edges w.r.t. ACR
7. Compute all useful combinations of $\lceil \frac{k}{2} \rceil$ optional edges using OSGRAPH
8. Take k reduced graph
9. Andrei-Chicco reduction (including loops) of k reduced graph
10. For original edges in reduced loop in ACR: try optional edge as switchover
11. Enumerate all spanning trees using Gabow and Myers' algorithm
12. Take spanning tree found
13. For original edges in reduced path in ACR: try optional edge as switchover
14. Deduce switchovers using spanning tree
15. Match switchover combinations from loops and spanning tree
All spanning trees tried?
no
yes
All edges decided or all k reductions executed?
no
yes
16. Determine which edges are unswitchable because of voltage or current capacities
17. Determine which edges need a switchover using precisely three or five switches
Output: list of seven results:

Result	**Features**
OUTPUT	list comprising of (for every edge in the initial spanning tree) the possible switchover edges (at most k) or the statement 'no switch possible', including the reason why
OPTIONAL_EDGES	array of edge numbers of optional edges (these edges do not need to be switchable)
BRANCHES	array of edge numbers of edges in branches (these edges are unswitchable)
BRIDGES	array of edge numbers of bridges (these edges are unswitchable)
UNSWITCHABLES_CAPACITIES	array of edge numbers of edges that are unswitchable because of voltage or current capacities (the load flow)
SWITCHABLES_3SWITCHES	array of edge numbers of edges that need precisely three switches in a switchover (there is a switchover using three switches and every other switchover contains at least three switches)
SWITCHABLES_5SWITCHES	array of edge numbers of edges that need five switches in a switchover

We elaborate on the steps of the overview:

- Step 1 is easy: all edges not present in the spanning tree TREE are optional edges. The optional edges do not need a switchover.
- Note that we do not accept loops in the first Andrei-Chicco reduction (in step 2) to avoid the computation of two many switchover combinations later on, as described in the previous subsection (subsection 6.2.4).
- Step 3 easily determines all edges in branches by taking all edges that have category zero after the ACR.
- Step 4 computes all switchovers using the reduced paths in the ACR. Note that there is not always an optional edge in a path that is reduced to a path edge in the ACR. Therefore, we only try to switch edges within a path using the optional edge if there actually is an optional edge. More specifically, for all reduced paths that contain an optional edge, for all original edges in the path, we try the optional

edge as switchover. Whether or not the switchover is accepted depends purely on the load flow computation, because the switchover is graph-theoretically correct.

- Note that we perform Schmidt's algorithm to find the bridges on the Andrei-Chicco reduced graph (in step 5). We translate the results back to edges in the original graph afterwards.

- Step 6 determines the ACR spanning tree G^A_{ACR} and the optional edges in G_{ACR}, as described in subsection 3.3.2.

- Step 7 selects all useful combinations of $\lceil \frac{k}{2} \rceil$ optional edges in preparation of the k reduction procedure. Subsection 3.3.1 described this principle to compute these useful combinations. Subsection 6.1.2 defined OSGRAPH, which is required for the selection of these useful combinations of optional edges.

- Step 10 computes all switchovers using the loops formed in the second ACR. In the path in the original graph corresponding to the loop there is precisely one optional edge. Thus, for each non-optional edge in the path we can try the optional edge in the path as switchover. More precisely, for all loops, for all original edges in the loop, we try the optional edge as switchover. Again, whether or not the switchover is accepted depends purely on the load flow computation, because the switchover is graph-theoretically correct.

- As may be clear, we compute the spanning trees of the graph after the second Andrei-Chicco reduction (in step 11).

- Step 13 computes all switchovers using the reduced paths in the second ACR. This is similar to step 4.

- To extract switchovers from new spanning trees found (in step 14), we use the symmetric difference of the initial spanning tree and the new spanning tree. We explained the initial strategy in section 2.3 and this method applied to an Andrei-Chicco reduction in subsection 3.3.2.

- Step 15 combines switchovers found in the actual k reduction by using loops, paths and/or new spanning trees. We explained this idea and its necessity at the end of subsection 3.3.2.

- We iterate the k reduction procedure until we have decided all edges or have computed all k reductions. We have an optional additional parameter in the $m-1$ algorithm, called 'accepted_undecided' that could change the acceptable ratio of undecided edges. For example, if we accept that we do not know for 1% of the edges whether they are switchable, we could define accepted_undecided = 0,01. Then we probably save computation time as we need less k reductions.

- In step 16, all edges that do not belong to the graph-theoretically unswitchable edges (the edges in branches and the bridges) and do not have a switchover are precisely the edges that are unswitchable because of the physical conditions. In other words, for all theoretical switchovers of these edges, at least one voltage or current capacity is exceeded. If we decided the output of all edges before the termination of all k reductions, then there are no unswitchable edges because of the load flow.

- Step 17 determines which edges are switchable but need more than one switch. We had already kept track of this during the execution of the algorithm so far.

- In a k reduction, it could happen that the second Andrei-Chicco reduction reduces to the empty graph. In such case all optional edges are in loops formed during the reduction process. In that case, we do not compute new spanning trees, for no new ones exist. In this particular case, we immediately go to step 15. Then we try to find new switchovers by combining switchovers from the different loops formed (there are no switchovers by using paths or spanning trees).

- The main output of the $m-1$ algorithm is OUTPUT, the list of edges together with a switchover or the reason why the edge does not have a switchover. The auxiliary function 'ShowOutput' displays the results of OUTPUT in a clear way.

- The output SWITCHABLES_5SWITCHES is omitted as output of the $m-1$ algorithm for the case $k = 3$.

6.3.2 The cases without load flow computation and $k = 1$

We present an overview of the $m - 1$ algorithms for the cases without load flow computation and $k = 1$. First note that in the case without load flow computation we can switch each edge using precisely one other edge or we cannot switch the edge at all. This follows from lemma 2.21 in section 2.3. Thus, although we do not assume that $k = 1$ in the case without the load flow, we can anyhow switch every edge with only one switch, or we cannot switch the edge at all.

The actual implementation of the two cases differs slightly, mainly regarding whether we perform a load flow computation before we determine a final possible switchover. Besides, step 11 below is omitted in the case without load flow computation. Further, we do not have EDGES_ORG, OSLIST and MSRLIST as input of the algorithm in that case and the result UNSWITCHABLES_CAPACITIES is not an output of the algorithm. However, the general structure of both variations of the algorithm is very similar.

On the next page, we display the overview of the $m - 1$ algorithms for the cases without load flow computation and $k = 1$. After that, we clarify some differences between this overview and the overview of the other cases in the previous subsection.

Main function to check $m-1$ principle: mainNoLoadFlow / maink1

$$\begin{tikzcd}
\textbf{Input: } \text{TRANSEDGES, EDGES\_ORG, OSLIST, MSRLIST, TREE, } n, m \arrow[d] \\
\text{1. Determine which edges are optional edges} \arrow[d] \\
\text{2. Andrei-Chicco reduction (including loops) of TRANSEDGES} \arrow[d] \\
\text{3. Determine which edges are in branches using ACR} \arrow[d] \\
\text{4. For original edges in reduced loop in ACR: try optional edge as switchover} \arrow[d] \\
\text{5. For original edges in reduced path in ACR: try optional edge as switchover} \arrow[d] \\
\text{6. Determine which edges are bridges using Schmidt's algorithm} \arrow[d] \\
\text{7. Make initial spanning tree and list of optional edges w.r.t. ACR} \arrow[d] \\
\text{8. Take } k \text{ reduced graph} \arrow[d] \\
\text{9. Andrei-Chicco reduction (including loops) of } k \text{ reduced graph} \arrow[d] \\
\text{10. For original edges in reduced loop in ACR: try optional edge as switchover} \arrow[d] \\
\textit{All edges decided or all } k \textit{ reductions executed?} \arrow[d, "yes"] \arrow[uuu, bend right=60, "no"'] \\
\text{11. Determine which edges are unswitchable because of voltage or current capacities} \arrow[d] \\
\textbf{Output: } \text{list of five results:}
\end{tikzcd}$$

Result	**Features**
OUTPUT	list comprising of (for every edge in initial tree) the possible switchover edge or the statement 'no switch possible', including the reason why
OPTIONAL_EDGES	array of edge numbers of optional edges (these edges do not need to be switchable)
BRANCHES	array of edge numbers of edges in branches (these edges are unswitchable)
BRIDGES	array of edge numbers of bridges (these edges are unswitchable)
UNSWITCHABLES_CAPACITIES	array of edge numbers of edges that are unswitchable because of the load flow

As all steps in this overview are present in the overview of cases $k = 3$ and $k = 5$, one can look at the explanatory notes in the previous subsection if needed. Note that this algorithm is a lot simpler than the previous.

In the cases $k = 3$ and $k = 5$, we first execute an Andrei-Chicco reduction without loops to avoid the computation of too many switchover combinations later on. However, we do not need to combine switchovers for the cases without load flow and $k = 1$, so we perform a complete Andrei-Chicco reduction in these cases (step 2 in the overview). Consequently, we can extract switchover information from the loops formed (in step 4).

Another difference is that we do not have OSGRAPH as input argument in the cases without load flow and $k = 1$. This is because we do not need to combine optional edges in the k reduction, for $k = 1$. In these cases, the k reduction procedure consists of adding the optional edges one at a time. Thus, there is no complication of searching for the required combinations, as we need all optional edges precisely once.

We now explain why we do not need spanning trees in the second overview, whereby steps 11 to 15 from the first overview all disappear. As a k reduction contains just one optional edge, the reduced graph contains just one cycle (for adding an edge to a spanning tree results in a graph having one cycle). If we execute the second Andrei-Chicco reduction on the k reduced graph, then the resulting graph is empty. Therefore, there are no real spanning trees of the graph and we skip all steps 11 to 15 from the first overview. As the cycle reduced to one loop during the second ACR, we can extract switchover information using this loop (step 10 in the second overview). This is the only step in each k reduction that brings about switchover information.

Last, note that in the case without load flow computation all edges really have an output at the end of the last k reduction. Strictly speaking, we thus never have to finish all k reductions completely in that case. However, it could happen that we have to terminate almost all k reductions, which is often the case in practice. For some of the edges may need a specific optional edge as switch and this optional edge could happen to be only in the last k reduction.

6.4 Practical comments

This section makes some remarks with respect to some technicalities of our code and the program R, so it is really about the implementation and not about any mathematics or logical reasoning. First of all, we should note that we have hitherto not given all implemented functions in our R code. We only presented the functions to prepare the data and all versions of the main function. However, we also implemented functions to compute the number of spanning trees using Kirchhoff's theorem (section 3.1, particularly theorem 3.8), to compute the Andrei-Chicco reduction (subsection 3.3.2), to compute all spanning trees using Gabow and Myers' algorithm (section 3.4), to compute the load flow (chapter 4, particularly section 4.3) and to compute all bridges using

Schmidt's algorithm (section 5.1). We assume one could understand those functions in R by reading the corresponding mathematical sections about the theory. We tried to stay as close as possible to the theory in the implementation in R. Moreover, we also implemented some other smaller auxiliary functions.

Second, we note that we only used the graph-theoretical package *igraph* (with extension *ggm*) in the data preparation functions. We explained this already in section 5.3. Consequently, in all other scripts and functions, we defined our own structures to represent graphs. Generally, we represented a graph as a set of edges (edge numbers) with given ends (nodes). More specific, a graph is represented as an $m \times 3$ matrix having as first column the edge number and as other two columns the ends of the edge.

We did so to stay as close as possible to the theory from the mathematical articles. This gave us a certain flexibility too, as we could implement every graph-theoretical auxiliary function ourselves, whilst *igraph* has only a limited number of functionalities. In practice, our own graph-theoretical approach turned out to be fast, whilst we experienced that some functions of the package *ggm* consumed a lot of time.

Furthermore, it is striking how slow R computes *for loops*. We save a lot of time by transforming for loops into vector (or matrix) definitions. We tried to do this if we could. Of course, there may be room for improvement in the code. However, for the time being it is already doing very well.

We finally note that R may not be the best computer program to compute graph-theoretically problems. There are definitely other computer programs that have more graph functionalities and have a better performance on these subjects. However, we programmed our algorithm in R because the majority of Alliander's models are implemented in R. As a result, one can add the $m - 1$ algorithm to these models.

Chapter 7

Results & Conclusion

This chapter presents the results of experiments with our implementation in R on different networks, including Allianders's entire MV network. We use the different versions of the $m-1$ algorithms (depending on k) as described in the previous chapter, in particular section 6.3. We display results for experiments both in a current as well as a future scenario. After displaying and explicating the results, we briefly conclude on the functioning of the algorithms.

7.1 Results

This section deals with the direct results of the $m-1$ algorithms in R on different MV networks. First, we test the algorithms on Alliander's entire MV network. We simulate the physical state of this network as it is today. Besides, we take some smaller MV networks and provide them with more marginal physical values. In this way, we simulate a future scenario of the network.

First, we consider the different networks used and we present the physical values chosen for each of the two scenarios. Second, we depict the results of the present scenario on the entire MV network. Last, we show the results of the future scenario on the different smaller MV networks. In this part we also go into further detail on the results' consistency for different algorithms.

As the networks that we will use are quite large, we will not present the specific output of one of the $m-1$ algorithms on such a network. However, to give insight in the shape of the output, we display a piece of ShowOutput(OUTPUT) (as in section 6.3) of some MV network. OUTPUT is the first return argument of each $m-1$ algorithm.

```
Edge 530: switch 604
Edge 531: switch not possible, it is in a branch
Edge 532: switch 458
Edge 533: switch not possible, because of voltage or current capacities
Edge 534: switch 429 and 455 and 1135
Edge 535: switch not possible, it is in a branch
Edge 536: switch 444 and 455 and 548
Edge 537: switch not possible, because of voltage or current capacities
Edge 538: switch 458
```

7.1.1 MV networks and two scenarios

This subsection presents the MV networks on which we test our $m - 1$ algorithms. In particular, it considers the entire MV network split into three parts and it defines some smaller test networks. Moreover, we provide the physical values chosen for both categories of networks. Hereby we distinguish two scenarios on the MV networks: a present scenario on the entire MV network and a future scenario on the smaller MV networks.

For both kinds of networks, we first present the factual information of the networks and hereafter the physical state (belonging to the scenario).

Alliander's entire MV network

Below, we display Alliander's MV network split into three parts, which are mutually disjoint, but together cover the entire MV network. These three parts are NORTH, WEST and EAST. The previous chapter (particularly subsection 6.1.2) already explained why we cut the MV network in these three parts.

The computation time of the data preparation (as in section 6.1) takes about five minutes for the entire network. In principle, one has to do this preparation only once. For one can still adjust the physical values afterwards. However, this relatively short preparation time also enables to always take the actual state of the network.

Network	**# Substation areas**	n	m	# links
NORTH	116	9300	10200	282
WEST	108	22500	25300	774
EAST	151	21400	24000	701
Total	375	53200	59500	1757

The new information in this table (with respect to the table presented in subsection 6.1.2) is firstly the number of substation areas in each of the parts, primarily provided for information purposes. These numbers follow from the substation names as on the GDSS website. Note that we consider both a control station ('regelstation') and switching station ('schakelstation') to be a substation (OS). As a result, the total number of substation areas may be above the expected number of substations, which equals about 350 substations.

The other additional information in this table is the number of MV links in each region. This number represents the number of edges that connect two substation areas at the MV level, i.e. the number of edges that lie in two substation areas. Due to the presence of these MV links we need to consider multiple substation areas at once to check the $m - 1$ principle, instead of considering the areas one by one. Such a MV link could provide a switchover solution for a possible broken edge by using two substation areas (explained in more detail in section 5.2). The number of MV links therefore indicates the degree of interconnectedness. The more the network is intertwined, the greater the likelihood of a switchover, but also the greater the computation time of the $m - 1$ algorithms. The latter is because the number of k reductions grows rapidly in that case.

Lastly we note that the number of nodes n equals the number of nodes after the merger of the HV/MV transformers (section 5.2). Hereby n is smaller than the number of nodes of the original network. As we merged all HV/MV transformers into one substation in each of the regions, the original number of nodes of the entire MV network equals $n_{\text{total}} + 372$.

Physical values for Alliander's MV network

For the given parts of the MV network we also need values of the physical quantities. As mentioned in section 6.1, these values are, unfortunately, not presented in our data. Therefore, we choose particular random values of these quantities to be able to test the $m-1$ algorithms on our data. Before we depict the chosen values, we would like to note that these values might not be entirely realistic. An electricity distribution grid expert would probably suggest other values. However, these values suffice to test the $m-1$ algorithms for our purposes. To that end, we need that not all switchovers are accepted by the load flow computation, but most of them are. In other words, we need that the load flow computation (more specific, the exceeding of a voltage or current capacity) limits the possible solutions, but it does not limit the solutions so far that there are no solutions at all for a lot of the edges.

As previously stated, we would like to simulate two scenarios: a state of the network comparable to how it is nowadays and a more unstable state in which a couple of the edges have no switchover that is accepted by the load flow. The latter state could represent a future state of the network.

Note that we do not test our algorithms on a state of the network where almost no edge has a changeover due to the physical conditions. We do not consider this situation to be very realistic. However, if one would like to test the algorithms on such a situation, the resulting computation time might be disappointing. This is because of the large number of switchover combinations the algorithms need to check. In the case $k = 1$, the additional computation time is most probably limited and in the case $k = 3$, the additional computation time is manageable. However, if $k \geq 5$, then the $m-1$ algorithms take probably too long. In this case one could only use this algorithm on small networks, i.e. on networks having at most 1000 edges.

Returning to the first scenario, we will provide the three parts of the entire MV network with physical values such that each edge will have at least one switchover (although by no means every switchover is accepted by the load flow). Thus, one could view NORTH, WEST and EAST including there physical values as Alliander's present networks. Each node and edge in the networks obtains the required physical value from an array of predetermined values, as can be seen in the table below. We allocate the values completely random.

Physical quantity	concerns	obtains random values from array
Power consumption	MSRs	(9000, 9500, 10000, 10000, 10500, 11000)
Minimum voltage capacity	MSRs	(9000, 9250, 9500, 9750, 10000)
Maximum voltage capacity	MSRs	(10750, 10800, 10850, 10900)
Voltage	OS's	(10250, 10500, 10750, 10750, 10750)
Current capacity	cables	(300, 350, 400)

Moreover, we define the impedance of a cable as 10^{-4} times its system length (the length of the cable). This system length (given in meters) is available in our data.

In anticipation of the second scenario, the smaller MV networks that we define below will represent a future state of the network. Their physical values will bring about a few dozen edges that no longer have a switchover.

Some smaller MV networks

The smaller test MV networks consist of all nodes and edges of some neighbouring substation areas, which we indicate below. Each succeeding network is an extension of the previous one. In this way, we tried to render the networks comparable. The number of edges ranges from several hundreds to several thousands.

Network	Substation areas	n	m	# links
SMALL1	AMO	128	136	0
SMALL2	AMO, DUK, KUN	373	410	3
SMALL3	AMO, DUK, KUN, TSD2, WYCH	651	728	22
SMALL4	AMO, DUK, KUN, TSD2, WYCH, WWGB, WWGC, VKG	1011	1135	42
SMALL5	AMO, DUK, KUN, TSD2, WYCH, WWGB, WWGC, VKG, TSD1, LEUT, WWGD	1503	1684	49
SMALL6	AMO, DUK, KUN, TSD2, WYCH, WWGB, WWGC, VKG, TSD1, LEUT, WWGD, ARN, SCA, DRT, WML	2101	2371	74
SMALL7	AMO, DUK, KUN, TSD2, WYCH, WWGB, WWGC, VKG, TSD1, LEUT, WWGD, ARN, SCA, DRT, WML, GB, BML1, BML2, DOD	2797	3158	84
SMALL8	AMO, DUK, KUN, TSD2, WYCH, WWGB, WWGC, VKG, TSD1, LEUT, WWGD, ARN, SCA, DRT, WML, GB, BML1, BML2, DOD, WLS, ELT1, ELT2, ELTSS, ZPRD, VST, TLML, TL2, CL2	3674	4137	127
SMALL9	AMO, DUK, KUN, TSD2, WYCH, WWGB, WWGC, VKG, TSD1, LEUT, WWGD, ARN, SCA, DRT, WML, GB, BML1, BML2, DOD, WLS, ELT1, ELT2, ELTSS, ZPRD, VST, TLML, TL2, CL2, CL1, BSD, BUU, TL3, TL1, LEUV, NRN	4651	5248	179

Note that the specific number of nodes, edges and links can differ over time. We take the actual state of the networks as given by IntellEvent, which includes the temporary shutdown of edges as a result of work or a breakdown. This generally concerns only a few cables, but it does yield small differences in the results.

Physical values for the smaller MV networks

We provide the cables in these networks with alternative values of the current capacity to let them be more susceptible to the exceeding of a capacity. Herewith, we force these networks in a more unstable state. We will see that a few dozen edges in these networks do not have a switchover because of the load flow condition. Note that the other physical quantities obtain values from the same arrays as the three parts of the entire MV network.

Networks	array of current capacity values
SMALL1, SMALL2, SMALL3, SMALL4, SMALL5	(30, 60, 80, 100, 120)
SMALL6, SMALL7	(45, 60, 80, 100, 120)
SMALL8, SMALL9	(50, 60, 80, 100, 120)

The reason that we do not provide each network with current capacity values from the same array is that we want the networks to have a comparable number of edges that do not have a switchover because of the load flow. The larger networks would have an enormous amount of such unswitchable edges otherwise, which is not the (realistic) case we want to consider. Conversely, the smaller networks do not have any such unswitchable edge if they obtain values from the array of the larger test networks.

Furthermore, note that we may assign different physical values to the same quantity to edges and nodes that are in several of the networks, because we randomly assign the physical values in each of the networks.

Before we present the results of the $m - 1$ algorithms on the networks given above, we make one more remark. The results of the algorithms on the networks are very dependent on the choices of networks (graph-theoretical properties) and physical values. Therefore, one could not generalize the results to every other part of Alliander's network. For example, one could not simply assume that the computation time is similar for two different networks with the same number of edges. The results on the given networks are, however, a good example of the behaviour of the algorithms.

7.1.2 Present scenario on Alliander's entire MV network

This subsection displays the results of the four versions of the $m - 1$ algorithm on the entire MV network, given physical values that represent a present state of the MV network. We first present and clarify the results regarding the computation time. Afterwards, we display and explain the relevant information sought, i.e. the number of edges in branches, bridges and physically unswitchable edges, among others.

Computation time of the $m - 1$ algorithms

Below, we display the computation time and relevant parameters, such as m and the number of load flow computations. Another relevant parameter is the number of k reductions. We indicate the number of k reductions used and the maximum number of k reductions for the given network, as well as the percentage of actually used k reductions.

without load flow

Network	n	m	# k reductions	comp. time
NORTH	9300	10200	728/732 (99,45 %)	1,5 min
WEST	22500	25300	2565/2640 (97,16 %)	14 min
EAST	21400	24000	2517/2535 (99,29 %)	13,5 min
Total	53200	59500	5810/5907 (98,36 %)	29 min

$k = 1$

Network	n	m	# k reductions	# load flows	comp. time
NORTH	9300	10200	730/732 (99,73 %)	7966	9 min
WEST	22500	25300	2640/2640 (100,0 %)	19284	54 min
EAST	21400	24000	2517/2535 (99,29 %)	17675	51 min
Total	53200	59500	5887/5907 (99,66 %)	44925	114 min

$k = 3$

Network	n	m	# k reductions	# load flows	comp. time
NORTH	9300	10200	7432/7796 (95,33 %)	7966	0,3 hours
WEST	22500	25300	50586/73865 (68,48 %)	19717	4,7 hours
EAST	21400	24000	36149/51131 (70,70 %)	17675	3,5 hours
Total	53200	59500	94167/132792 (70,91 %)	45358	8,5 hours

$k = 5$

Network	n	m	# k reductions	# load flows	comp. time
NORTH	9300	10200	44661/75432 (59,21 %)	8019	85 min

For WEST, after 12 hours we have computed 149850/2164671 k reductions (6,923 %). At this stage we only need to decide 7 edges (0,0276 %).

For EAST, after 37 hours we have computed 467840/1033798 k reductions (45,25 %). At this stage we only need to decide 3 edges (0,0125 %).

Observations regarding the algorithms' computation time

Clearly, the computation time grows rapidly if we add the load flow computation in the $m - 1$ algorithm, as well as if we subsequently increase k. In the case $k = 5$, the computation time increases so fast that we did not finish the computation for WEST and EAST. Most probably, the $m - 1$ algorithm terminates in those cases only after a couple of days. However, we could accept that a few edges are not yet decided, i.e. we do not know whether they have an acceptable switchover by the load flow or not. If we do so, then the algorithm terminates within half a day for WEST and EAST in the case $k = 5$. For example, if we accept that 0,05 % of the edges is undecided, then we know to what extend the $m - 1$ principles holds for 99,95 % of the edges within 12 hours of computation.

Comparing the cases without load flow computation and $k = 1$, it is remarkable that the addition of the load flow slows down the algorithm with a factor four. These two $m - 1$ algorithms do not differ except for the presence of the load flow computation at all places in the algorithm where we found a switchover solution. Note that the number of k reductions is similar for both cases. Thus, the load flow computation takes up a large part of the computation time. In general, the classical load flow is known to cost a lot of computing time. Apparently, the linear load flow model that we use still takes a lot of time, although it is much better than the classical model.

Comparing the cases $k = 1$ and $k = 3$, we see that the number of load flow computations does not differ that much. Therefore, it is likely that the number of k reductions causes the significant growth of the computation time. This is conceivable, because there are some important steps in each k reduction. In particular, in each k reduction we perform an Andrei-Chicco reduction, search for spanning trees and search for switchovers in different manners. Thus, the computation of these steps takes a lot of time eventually, if we have to perform many k reductions. The same holds when comparing the cases $k = 3$ and $k = 5$. The growth factor from the case $k = 1$ to $k = 3$ equals about four. This is the same growth as from the case without load flow to $k = 1$, but this is a coincidence.

Note that we limited the k reductions to only useful combinations of optional edges in neighbouring substation areas. Therefore, we decreased the number of k reductions a lot, by almost a factor of 1000. Unfortunately, the number of k reductions is still quite high. Up to and including the case $k = 3$ the computation time for all k reductions is manageable for the entire network.

Last, we look at the number of k reductions that is really used. In the cases without load flow and $k = 1$, we need almost all k reductions. This is because we add each optional edge in only one k reduction, but some edges may need a specific optional edge as switchover. Thus, it is likely that we have to finish almost all k reductions to come across these ‘necessary’ optional edges. In the cases $k = 3$ and $k = 5$, we do not need all k reductions if there are no physically unswitchable edges (edges that have no switchover because of the load flow), for the majority of the optional edges is in multiple of the k reductions. This fortunately saves a large part of the (maximum) computation time.

Outcomes of the $m-1$ algorithms

We present some interesting numbers that result from the $m-1$ algorithms on the entire MV network. As there are a lot of edges, we do not depict the specific switchovers for the edges that have one or the specific edge numbers of edges that do not have a switchover for a particular reason. Note that each successive $m-1$ algorithm provides more information than the previous one. Therefore, we only present the information the first time. For example, each $m-1$ algorithm determines which edges are bridges, but we only note the numbers of bridges in the table of the case without load flow computation.

without load flow

Network	**# edges in branches**	**# other bridges**	**Total # unswitchables**
NORTH	1348 (13,25 %)	20 (0,1966 %)	1368 (13,45 %)
WEST	3204 (12,63 %)	57 (0,2247 %)	3261 (12,86 %)
EAST	3667 (15,27 %)	31 (0,1291 %)	3698 (15,40 %)
Total	8219 (13,80 %)	108 (0,1814 %)	8327 (13,98 %)

The unswitchable edges are the mathematically unswitchable edges in this case, because we did not add the physical values and load flow computation yet. Note that we could call these edges also graph-theoretical unswitchable edges and that these edges are precisely all bridges of the network.

$k = 1$

Network	**# math. unswitchables**	**# phys. unswitchables**	**Total**
NORTH	1368 (13,45 %)	0 (0 %)	1368 (13,45 %)
WEST	3261 (12,86 %)	1 (0,003943 %)	3262 (12,86 %)
EAST	3698 (15,40 %)	0 (0 %)	3698 (15,40 %)
Total	8327 (13,98 %)	1 (0,001679 %)	8328 (13,98 %)

The physically unswitchable edges are the edges that have no switchover because of the load flow computation, i.e. because of exceeding a voltage or current capacity for each graph-theoretically possible switchover.

$k = 3$

Network	**# 3 switches**	**# phys. unswitchables**	**Total # unswitchables**
NORTH	0 (0 %)	0 (0 %)	1368 (13,45 %)
WEST	1 (0,003943 %)	0 (0 %)	3261 (12,86 %)
EAST	0 (0 %)	0 (0 %)	3698 (15,40 %)
Total	1 (0,003943 %)	0 (0 %)	8327 (13,98 %)

'3 switches' represents the edges that are switchable, but need at least three switches in a switchover.

Clearly, all graph-theoretically switchable edges have a switchover. All but one even have a switchover using only one switch. This does not mean that each switchover is accepted by the load flow, but there is at least one that is accepted for each graph-theoretically switchable edge.

Furthermore, note that the outcomes of the case $k = 5$ equal the outcomes for the case $k = 3$, because there are no physically unswitchable edges in the case $k = 3$ already. Therefore, we did not display outcomes of the case $k = 5$ explicitly.

Consequently, we did not even need to compute the $m - 1$ algorithm for the case $k = 5$ after we computed the case $k = 3$. However, we only know this fact after the computation of the case $k = 3$.

Last, we note once more that these results depend on the specific physical values chosen. A comparable network that we gave just a little different distribution of the physical values could have different results. Thus, one should not generalize these results without looking at the chosen physical values. Of course, the results of the $m - 1$ algorithm without load flow computation are not dependent on the physical values. They are, however, dependent on the specific graph-theoretical shape they have, which could change a bit if the state of the network changes (given by IntellEvent).

7.1.3 Future scenario on example MV networks

This subsection presents the results of the $m - 1$ algorithms for different values of k on the given smaller MV networks (in section 7.1.1). We assume the physical values as given in that section. These values represent a future state of the MV network. First, we show the results on these smaller networks regarding the computation time. Second, we plot the coherence of the relevant parameters and the computation time. Last, we display some of the information that results from the $m - 1$ algorithms, for example the number of mathematically or physically unswitchable edges.

We do not present the results of the $m - 1$ algorithm without the load flow computation, because this algorithm functions already very well on the entire MV network. Therefore, we predict that this function will work even faster on smaller networks.

Computation time of the $m - 1$ algorithms

We display the computation time and relevant parameters for all example networks and $m - 1$ algorithms indicated below. Note that we just show one number of k reductions. As we chose the current capacity values in such a way that there is at least one edge physically unswitchable, we need to execute all k reductions. Thus, the number of k reductions used equals the maximum number of k reductions.

$k = 1$

Network	n	m	**# k reductions**	**# load flows**	**comp. time**
SMALL1	128	136	5	74	1 sec
SMALL2	373	410	31	299	5 sec
SMALL3	651	728	70	605	11 sec
SMALL4	1011	1135	118	973	18 sec
SMALL5	1503	1684	171	1410	29 sec
SMALL6	2101	2371	260	1912	51 sec
SMALL7	2797	3158	349	2517	85 sec
SMALL8	3674	4137	448	3173	111 sec
SMALL9	4651	5248	576	4061	153 sec

$k = 3$

Network	n	m	**# k reductions**	**# load flows**	**comp. time**
SMALL1	128	136	36	117	0,03 min
SMALL2	373	410	292	2438	0,68 min
SMALL3	651	728	1095	8476	2,45 min
SMALL4	1011	1135	2072	9276	2,85 min
SMALL5	1503	1684	2859	10132	4,20 min
SMALL6	2101	2371	4615	16374	7,35 min
SMALL7	2797	3158	6146	14944	8,27 min
SMALL8	3674	4137	7762	10927	9,88 min
SMALL9	4651	5248	10483	12539	15,38 min

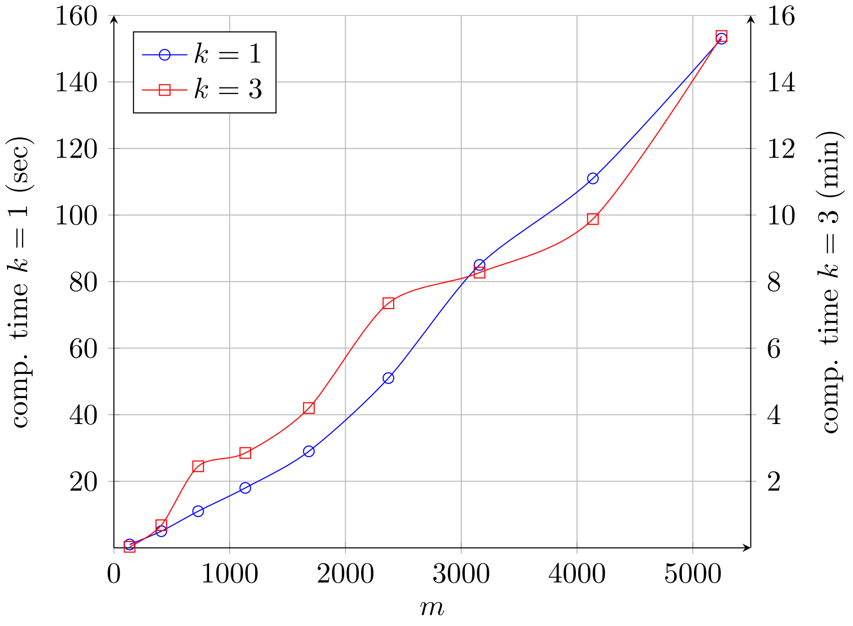
$k = 5$

Network	n	m	**# k reductions**	**# load flows**	**comp. time**
SMALL1	128	136	84	492	0,002 hours
SMALL2	373	410	1903	61967	0,24 hours
SMALL3	651	728	12562	921031	3,76 hours
SMALL4	1011	1135	30951	197526	1,14 hours
SMALL5	1503	1684	41093	285919	1,77 hours
SMALL6	2101	2371	75957	1535102	8,98 hours
SMALL7	2797	3158	94346	595263	4,89 hours
SMALL8	3674	4137	124629	539043	5,44 hours
SMALL9	4651	5248	181355	644348	9,22 hours

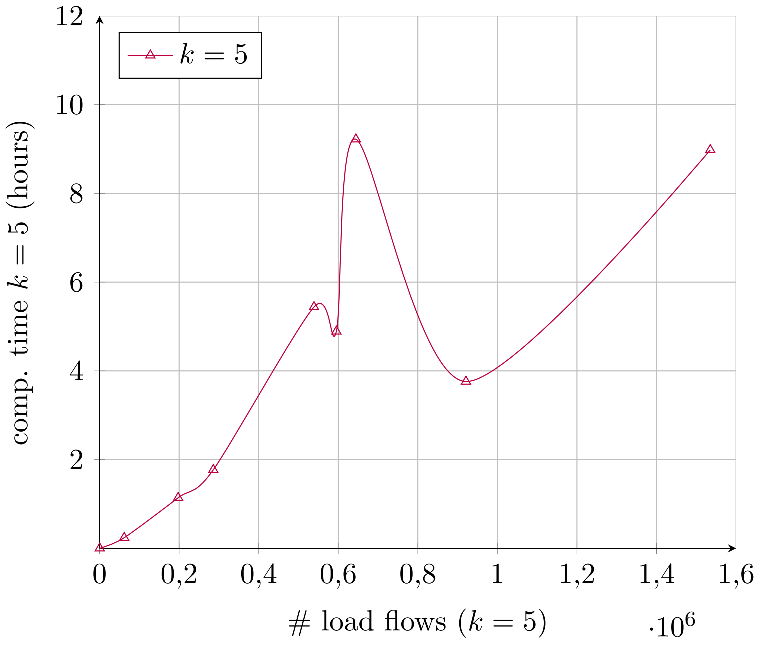
Observations regarding the computation time of the algorithms

In this future scenario, the computation time grows rapidly if we increase k, just as we saw in the results of the present scenario on the entire MV network in the previous subsection. A difference in the results between this scenario and the previous one is that here both the number of k reductions and the number of load flow computations grows rapidly if we increase k. In the previous subsection, mainly the number of k reductions grew. The additional growth of the number of load flow computations probably explains the growth of the computation time by a factor of about 5,5 to 9 instead of a factor 4 from the case $k = 1$ to the case $k = 3$.

Below, we show some plots of the results regarding the computation time. We explain some noteworthy facts after each chart.

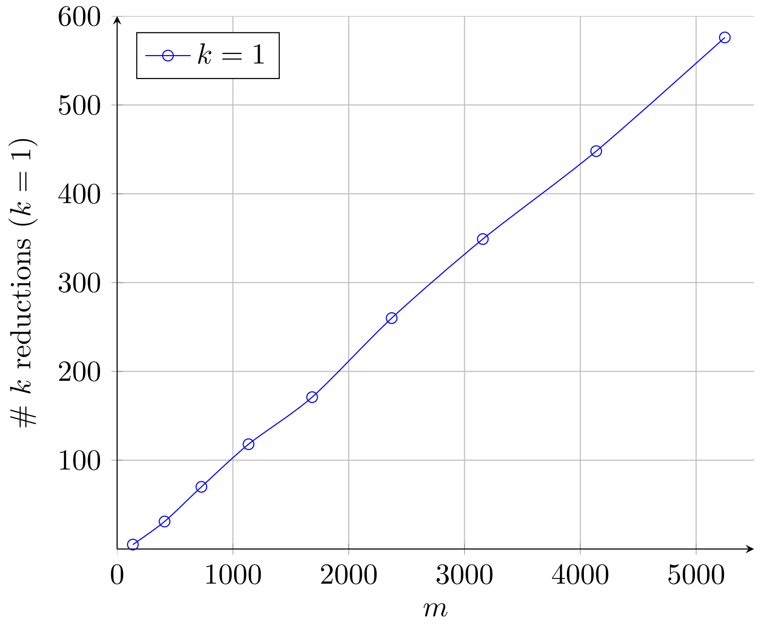


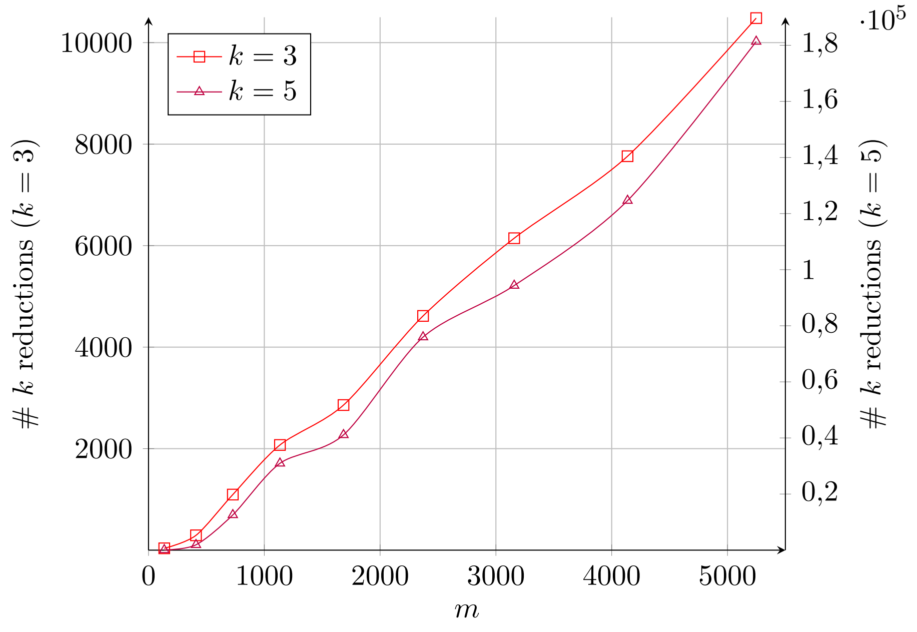
We see that the computation time expressed as depend variable of the number of edges m grows about linearly in both the cases $k = 1$ and $k = 3$. However, the computation time in the case $k = 3$ is about 6 times higher than in the case $k = 1$. If we compare the computation time in the case $k = 5$ too (in the table on the previous page), then we see that the computation time in that case fluctuates a lot. Thus, in the case $k = 5$ there is not a clear correlation between m and the computation time. We can probably explain this by the major fluctuation of the number of load flow computations. If we sort the number of load flow computations for this case and depict the corresponding computation time, then we have the following figure:



In the case $k = 5$, the computation time seems particularly dependent of the number of load flow computations. The plot is partly linear, but there are some deviating points. Probably, for these points, m or the number of k reductions has some effect on the computation time. In particular, the deviating peak in the middle of the chart belongs to the results of SMALL9, which is the network that has the most edges and the most k reductions. Probably, these numbers have some influence too. On the other hand, it is also true that each load flow computation itself takes a bit more time for a larger network, which can ultimately explain the discrepancy.

The reason why the number of load flow computations is quite fluctuating is not immediately clear. This number is probably very dependent on the graph-theoretical structure and the possible switchovers that occur but are unfortunately unsuitable due to the load flow. If there are a lot of such edges, the number of load flow computations increases rapidly. We will elaborate on this at the end of this subsection.

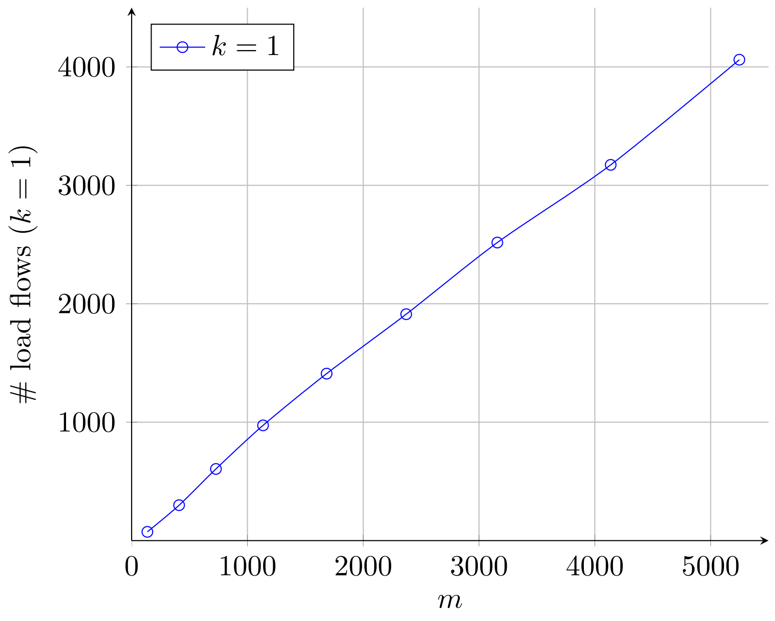


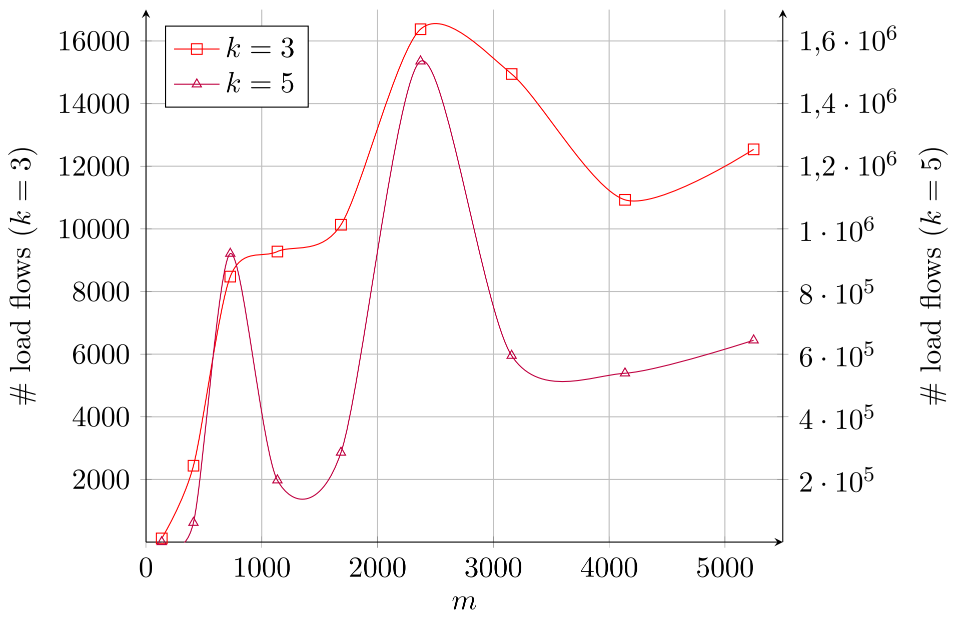


The above two figures show the number of k reductions in the cases $k = 1$, $k = 3$ and $k = 5$ as function of m. Clearly, the relationships are fairly linear. In the case $k = 1$, we can clarify this. In that case, the number of k reductions equals the number of optional edges of the first Andrei-Chicco reduced graph. In electricity distribution grids, the number of optional edges always represents approximately the same proportion of the total number of edges, so the number of k reductions grows linearly with the number of edges m.

In the case $k = 3$ and $k = 5$, this linear growth was not expected. Initially, we took all combinations of two or three optional edges as k reductions, making the number of k reductions grow quadratically and cubically with m. However, we lowered the number of k reductions by taking only all necessary combinations in neighbouring substation areas.

Consequently, it is no longer easy to see how the number of k reductions increases when m grows. However, the chart shows that they grow almost linearly. The small deviations are similar for $k = 3$ and $k = 5$. We can probably explain this by the specific shapes of the networks (graph-theoretically).





These last plots show the number of load computations dependent on m for the all cases of k. The relationship is clearly linear in the case $k = 1$, but it is erratic in the other cases. We do witness some correlation between the cases $k = 3$ and $k = 5$, which probably results from the specific structure and physical values that each selected network has.

We can explain the linear behaviour in the case $k = 1$ by the lack of the computation of switchover combinations. Those are present in the cases $k = 3$ and $k = 5$, but the number is unpredictable because it depends on the number of undecided edges at that point in the $m - 1$ algorithm. In the case $k = 1$, a lot of the edges have just one possible switchover, following from a loop or a path, that cannot be combined with other switchovers. Therefore, the number of load flow computation is even less than m, for the graph-theoretically unswitchable edges cause no load flow computation and most of the other edges get just one load flow computation.

Outcomes of the $m - 1$ algorithms

We display some numbers that result from the $m - 1$ algorithm on the smaller MV networks. Note again that each successive $m - 1$ algorithm provides more information than the previous one. Therefore, we display the information the first time only.

$k = 1$

Network	**# math. unswitchables**	**# phys. unswitchables**	**Total**
SMALL1	55 (40,44 %)	1 (0,7353 %)	56 (41,18 %)
SMALL2	89 (21,71 %)	26 (6,341 %)	115 (28,05 %)
SMALL3	123 (16,90 %)	41 (5,632 %)	164 (22,53 %)
SMALL4	135 (11,89 %)	22 (1,938 %)	157 (13,83 %)
SMALL5	231 (13,72 %)	42 (2,494 %)	273 (16,21 %)
SMALL6	305 (12,86 %)	13 (0,5483 %)	318 (13,41 %)
SMALL7	444 (14,06 %)	19 (0,6016 %)	463 (14,66 %)
SMALL8	591 (14,29 %)	16 (0,3868 %)	607 (14,67 %)
SMALL9	724 (13,80 %)	24 (0,4573 %)	748 (14,25 %)

The mathematically unswitchable edges are the graph-theoretically unswitchable edges, these are precisely all bridges of the network.

An important note in the cases of smaller networks is that some edges may seem mathematically unswitchable here, although there are not in practice. This is because we cut the network in the desired part containing only particular substation areas. However, some edges may be connected to other nodes in other substation areas. These edges may therefore be wrongly regarded as unswitchable.

If we take a look at the percentages of mathematically unswitchable edges, then we notice that this percentage drops for the larger test networks. In practice, the networks generally have a comparable number of mathematically unswitchable edges, at least within each geographical region. These deviating percentages arise from unjustly declared mathematically unswitchable edges.

Furthermore, we note that the chosen physical values caused a few dozen physically unswitchable edges in almost all example MV networks.

$k = 3$

Network	**# 3 switches**	**# phys. unswitchables**	**Total # unswitchables**
SMALL1	0 (0 %)	1 (0,7353 %)	56 (41,18 %)
SMALL2	9 (2,195 %)	17 (4,146 %)	106 (25,85 %)
SMALL3	16 (2,198 %)	25 (3,434 %)	148 (20,33 %)
SMALL4	14 (1,233 %)	8 (0,7048 %)	143 (12,60 %)
SMALL5	27 (1,603 %)	15 (0,8907 %)	246 (14,61 %)
SMALL6	11 (0,4639 %)	2 (0,08435 %)	307 (12,95 %)
SMALL7	17 (0,5383 %)	2 (0,06333 %)	446 (14,12 %)
SMALL8	8 (0,1934 %)	8 (0,1934 %)	599 (14,48 %)
SMALL9	12 (0,2287 %)	12 (0,2287 %)	736 (14,02 %)

Comparing the results for $k = 1$ and $k = 3$, we see that some of the physically unswitchable edges in the case $k = 1$ have a switchover using three switches in the case $k = 3$. The number of physically unswitchable edges will decrease if we increase k.

Moreover, for most example networks the number of physically unswitchable edges at least halves from the case $k = 1$ to the case $k = 3$. Thus, we see that in this future scenario, where a couple of the edges were not physically switchable in the case $k = 1$, a lot of them do have a switchover in the case $k = 3$. Note that this does not need to be the case if there were a lot of physically unswitchable edges, for the capacity values may then be that marginal that these edges also do not have a switchover in the case $k = 3$.

$k = 5$

Network	**# 5 switches**	**# phys. unswitchables**	**Total # unswitchables**
SMALL1	0 (0 %)	1 (0,7353 %)	56 (41,18 %)
SMALL2	0 (0 %)	17 (4,146 %)	106 (25,85 %)
SMALL3	0 (0 %)	25 (3,434 %)	148 (20,33 %)
SMALL4	3 (0,2643 %)	5 (0,4405 %)	140 (12,33 %)
SMALL5	1 (0,05938 %)	14 (0,8314 %)	245 (14,55 %)
SMALL6	0 (0 %)	2 (0,08435 %)	307 (12,95 %)
SMALL7	0 (0 %)	2 (0,06333 %)	446 (14,12 %)
SMALL8	0 (0 %)	8 (0,1934 %)	599 (14,48 %)
SMALL9	0 (0 %)	12 (0,2287 %)	736 (14,02 %)

Now comparing the results for $k = 3$ and $k = 5$, we observe that some of the physically unswitchable edges in the case $k = 3$ have a switchover using five switches in the case $k = 5$, but there are not so many.

7.2 Conclusion

Based on the results of the previous section, we conclude the following:

> *Using our $m-1$ algorithms we can efficiently check the $m-1$ principle on Alliander's MV networks. In particular, in the cases without load flow computation, $k=1$ and $k=3$, the $m-1$ algorithm is sufficiently fast to check the $m-1$ algorithm for Alliander's entire MV network of about 60.000 cables. It takes up to half an hour, two hours and 8,5 hours, respectively. In the case $k=5$, we could check the $m-1$ principle for an MV network having several thousand cables within a couple of hours.*

First of all, this main result is a great improvement of the computation time that the $m-1$ computation takes using the tool *Vision*. Vision can only do the computation in a few substation areas at once, having only several hundreds (instead of thousands) of cables. Section 1.3 described the $m-1$ computation in Vision on an example network consisting of 136 cables, which equals MV network SMALL1 in our implementation (subsection 7.1.1). It took Vision 33 minutes to complete this computation, whilst our $m-1$ algorithm needed only 7 seconds (both in the case $k=5$).

Second, note that the results of the $m-1$ algorithms are dependent on the physical values of the edges and nodes. It is important that these values are added to the data, yielding the m-1 algorithms ready to use and allowing final analyses to be done.

In particular, our $m-1$ algorithms are useful for physical values corresponding to both a present state of the MV networks and a future state of the networks, as we showed in the two different scenarios in the previous section. However, note that the computation time is shorter on a present scenario, because there are less switchover solutions rejected by the load flow computation.

Furthermore, we note that we have better results than presented above if we accept a small percentage of edges to have no result, i.e. we do not know whether they have a switchover or not. This is especially useful in the case $k=5$. If we accept that 0,05 % of the edges is 'undecided', then we can check the $m-1$ principle on larger networks having up to 15.000 cables within a couple of hours. The small number of undecided edges can then be determined 'manually', for example by using Alliander's control plan ('bedieningsplan'), which generates switchover solutions for a particular edge.

On the other hand, there may be a few ways to improve the computation time of the $m-1$ algorithms, particularly in the case $k=5$. We will suggest some improvements in the next chapter.

Chapter 8

Discussion & Future Work

This chapter discusses the usefulness and some possible improvements of the $m-1$ algorithms. On top, it suggests extensions of the m-1 algorithms to related problems as well as generalisations to related research areas. Both may present fruitful avenues for future research.

First, we would like to repeat that the $m-1$ algorithms as presented in section 6.3, which solve the $m-1$ problem, are a great improvement of the existing $m-1$ computation in the program *Vision*. Not only are the algorithms developed in this thesis much faster than the implementation in Vision, they are also more general than the method in Vision. Indeed, it allows us to check the $m-1$ principle on any combination of substation areas and in particular on Alliander's entire MV network. Vision could only perform this computation on a few areas instead of hundred at once. Note that the $m-1$ algorithm for the case $k=5$, which is the slowest performing algorithm, is still much faster than the implementation in Vision.

By contrast, there are always improvements possible. We present some suggestions to improve the $m-1$ algorithms below. Afterwards, we mention some generalisations of the algorithms and theory presented in this thesis. Using these, one can apply some of the findings of this project to other problems too.

8.1 Improvements of the $m-1$ algorithms

Despite the positive results of this project, as mentioned above and in section 7.2, we would like to make a few comments on the algorithms as they are now.

First of all, the computation time of the $m-1$ algorithm for the case $k=5$ on Alliander's entire MV network is yet too long to repeat the computation weekly, for example. Therefore, there is a need for further reducing the computation time in this case. Of course, one could improve the implementation in R itself by using the most efficient declarations in R, as was partly presented in section 6.4. We will, however, suggest improvements based on the general implementation at a higher level, which we will present at the bottom of the next page.

Besides improving the computation time, we note two points of attention concerning the physical state of the MV networks. First, one has to add the true physical values of each node (MSR and OS) and cable. Doing so would allow the $m-1$ algorithms to return realistic results and render the algorithms ready to use. Second, the implemented algorithms use static physical values for each call of such an algorithm. However, the physical values for the edges and nodes may differ over a day or over years, for consumers of course do not use the same amount of electricity constantly. It would probably be better if we take into account such a physical profile into the $m-1$ algorithms. An electricity distribution expert could possibly define how to implement this in the algorithms. Nevertheless, using the current algorithms, one could repeatedly use an $m-1$ algorithm on different physical values and extract relevant information about the variation already.

At the boundary of the computation time and the physical values, we notice another issue. The computation time increases if the number of physical unswitchable edges increases. This lessens the capability of the $m-1$ algorithm for the case $k=5$ on future scenarios on large networks (≥ 1000 cables) having a poor physical state. As this could also be an area of application of the algorithms, this requires further improvement regarding the computation time.

Furthermore, recall that the initial configuration of the network is not a spanning tree in most cases, but we assume a configuration to be one. This caused the adaptation of the initial configuration as explained in section 5.3. This spanning tree may, however, not be the most realistic state of the network. Therefore, it would be better if an expert would take a look at the initial configuration and would turn it into the most realistic spanning tree (potentially adjusting some physical values on the way).

We list different enhancements regarding the $m-1$ algorithms' computation time:

- Subsection 3.3.1 explained the limitation to only useful combinations of optional edges to form the k reductions, which improved the total number of k reductions a lot. However, we still test k reductions that are not useful, depending on the specific structure within a substation area. Subsequently, it would be useful if we select all useful combinations of optional edges beforehand. As this requires to observe the specific local graph structure within a substation area (or within two neighbouring substation areas), it probably takes quite a while. However, we only have to perform this preparation once (as long as the network does not change). Subsequently, we can use this list of suitable combinations of optional edges (depending on k) as input to the corresponding $m-1$ algorithm. This definitely decreases the computation time in the cases $k=3$ and $k=5$, for we saw in subsection 7.1.3 that the growth of the number of k reductions has a negative impact on the computation time.

- As the number of load flow computations has a substantial impact on the computation time too (as we saw in 7.1.3), it would be beneficial if we decrease the computation time of the load flow computation. As the linear load flow is already fast, it is probably hard to speed up the computation itself. However, we compute the load flow over the entire new spanning tree, whilst only some edges (at most $k+1$) differ from the initial spanning tree. Therefore, it is sufficient to compute the load flow only in the substation areas where a change has occured. This likely saves computation time.

- We could implement the method of *parallel computing*, i.e. computing different parts of the algorithm simultaneously on different processors. In our case, we could try to run different k reductions concurrently. We have to take care of the assignment of new switchovers in OUTPUT, but except for this the k reductions are independent. We possibly compute more switchovers overall if we parallelise the k reductions, because of the possibility to compute switchovers for the same edge in different k reductions. However, by the simultaneous computations we save a lot of computation time in the end.

8.2 Generalisations and extensions of the $m-1$ algorithms

Now that we have found well-functioning $m-1$ algorithms, we could use some (concepts used in the) algorithms on related problems too. Therefore, we present some generalisations to other research areas and some extensions of the $m-1$ algorithms:

- Note that the results of the $m-1$ algorithms could also be useful in a malfunction situation, for the results display a switchover for the failure edge if it exists. However, in a malfunction situation we only need a switchover for a specific edge, whilst the $m-1$ algorithms search for switchovers for every graph-theoretically switchable edge. In such case, it is probably faster to use an algorithm that searches for a specific switchover locally. Nonetheless, some of the mathematical theory of this thesis may be of interest to such an algorithm. For example, the enumeration of all spanning trees in section 3.4 or the listing of all bridges in section 5.1 may prove relevant for other algorithms. The reduction methods can also be useful.

- This thesis only took Alliander's MV networks into consideration. However, with some modifications we could also use the $m-1$ algorithms on other types of networks where a configuration of the network is radial (a spanning tree) and the question whether an edge is switchable is relevant.

  In the first place, the low voltage networks are examples of such networks. If we consider each MSR in a low voltage network as slack bus (an OS in the MV network) and the other nodes as load buses, the properties of the low voltage network are similar to those of an MV network. We have adapted our $m-1$ algorithm for the case without load flow computation to the low voltage case too. In this way, one can check the $m-1$ principle for Alliander's low voltage networks.

  Besides electricity distribution networks, the $m-1$ algorithm without load flow computation could be useful for other kinds of networks. Potentially, the algorithm is of interest for networks of pipelines, for example water or gas pipeline networks. The same holds for underground fiber optic or telephone cable networks.

Note that if the size of another network differs a lot from Alliander's MV networks, it could be of use to revise the order of the reductions and algorithms. The considerations presented in section 6.2 could help to make such a decision or one can take the relevant theory from chapters 3 and 5.

- Although we only considered the cases $k \leq 6$, the theory presented is also applicable to larger values of k. However, this may slow down the $m-1$ algorithm and it may not be feasible to apply such an $m-1$ algorithm on very large networks.

- The theory required to define the $m-1$ algorithms can also be used to find $m-2$ algorithms. An MV network satisfies the $m-2$ principle if the network has a possible reconfiguration for any pair of broken cables, i.e. the $m-2$ problem consists of finding switchovers for combinations of two 'broken' cables. We can realise an $m-2$ algorithm in the following manner: if T_j is a new spanning tree found, we can take two edges in $G^A \setminus T_j$. The symmetric difference $G^A \triangle T_j$ without the two edges forms a switchover for the combination of the two 'broken' edges. Using this idea, it is not hard to transform an $m-1$ algorithm to an $m-2$ algorithm. However, as there are many more combinations of two edges than single edges, the output of the algorithm is larger than in the $m-1$ case and the computation time increases too. An interesting fact is that the high voltage networks should fulfill this $m-2$ principle. Henceforth, an $m-2$ algorithm is relevant for the high voltage networks.

  More generally, one could be interested in the fulfillment of the $m-c$ principle, where $c \leq m$ is a positive integer. We could adapt an $m-1$ algorithm to an $m-c$ algorithm too, but such an algorithm may turn out impracticable due to the large number of combinations of 'broken' edges.

- Instead of looking at one or more broken edges, one could also be interested in broken nodes (MSRs or OS's). We call this problem of determining a switchover for each possible 'broken' node the $n-1$ *problem*. If a node breaks down, then one needs to switch all edges connected to that node. Therefore, the $n-1$ problem relates to the $m-c$ problems. With some more adjustments, an $m-1$ algorithm could also be converted into an $n-1$ algorithm.

- We come back to the remark at the beginning of section 6.3 concerning *optimising* the switchovers or not. In the present $m-1$ algorithms, we only optimise the switchovers in the way that they consist of as few switches as possible. However, we could adapt the algorithms to make them search for the best switchover (if it exists) for each edge, using some predefined evaluation criteria. Note that this would increase the computation time.

To balance between optimisation and the computation time, we could also save and return multiple switchovers per edge, thereby providing the user some choice between switchovers.

- Some of the mathematical theory in this thesis, in particular the theory in chapters 3 and 5, is also useful for other electricity distribution grid problems. The configuration of such networks is always a spanning tree. Hence, if one searches for all allowed configurations, the spanning tree listing in section 3.4 is applicable. To avoid to many spanning trees (as shown in section 3.2), a reduction method as in section 3.3 may be necessary. Similarly, Schmidt's bridge algorithm may provide useful (section 5.1) as well as the merger of the HV/MV transformers (section 5.2).

  Examples of electricity distribution network issues include finding the best configuration with respect to the power loss through the cables (as in Van der Meulen (2015) [20]) or with respect to the *outage consumer minutes* (defined in section 1.3).

- Last, note that the preparation function as presented in subsection 6.1.2 can also be used for other purposes. For example, it could find the components and cycles of a network, which may be of interest for other network problems. In the case of the low voltage network, we adapted the preparation functions such that they returned these graph structures, because they were not yet known.

Some of the above considerations may provide fruitful avenues for future research.

Bibliography

[1] J.A. Bondy, U.S.R. Murty, *Graph Theory*, Graduate Texts in Mathematics, Springer, 2008.

[2] M. Sipser, *Introduction to the Theory of Computation*, Thomson Course Technology, 2006.

[3] R.P. Stanley, *Topics in Algebraic Combinatorics*, book for course *Algebraic Combinatorics*, Harvard University, 2000, version of 1 february 2013.

[4] M. Marcus, H. Minc, *Introduction to Linear Algebra*, MacMillan, 1965.

[5] Z.R. Bogdanowicz, *Undirected Simple Connected Graphs with Minimum Number of Spanning Trees*, Discrete Mathematics, Vol. 309, No. 10, 2009.

[6] H. Andrei, G. Chicco, *Identification of the Radial Configurations Extracted From the Weakly Meshed Structures of Electrical Distribution Systems*, IEEE Transactions on Circuits and Systems, Vol. 55, No. 4, 2008.

[7] H.N. Gabow, E.W. Myers, *Finding all Spanning Trees of Directed and Underícted Graphs*, SIAM Journal on Computing, Vol. 7, No. 3, 1978.

[8] T. Matsui, *An Algorithm for Generating All the Spanning Trees in Undirected Graphs*, Technical Report METR 93-08, Dept. of Mathematical Engineering and Information Physics, University of Tokyo, 1993.

[9] S. Kapoor, H. Ramesh, *Algorithms for enumerating all spanning trees of undirected and weighted graphs*, SIAM Journal on Computing, Vol. 24, No. 2, l995.

[10] A. Shioura, A. Tamura, *Efficiently Scanning All Spanning Trees of an Undirected Graph*, Journal of the Operation Research Society of Japan, Vol. 38, No. 3, l995.

[11] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974.

[12] J.M. Schmidt, *A Simple Test on 2-Vertex- and 2-Edge-Connectivity*, lecture notes for course *Advanced Graph Algorithms*, Max Planck Institute for Informatics, 2012.

[13] R.E. Tarjan, *Depth-first Search and Linear Graph Algorithms*, SIAM Journal on Computing, Vol. 1, No. 2, 1972.

[14] H.N. Gabow, *Path-based Depth-first Search for Strong and Biconnected Components*, Information Processing Letters, Vol. 74, No. 3, 2000.

[15] R. Diestel, *Graph Theory*, Graduate Texts in Mathematics, Springer, 2010.

[16] P. van Oirsouw, *Netten voor Distributie van Elektriciteit*, Phase to Phase, 2012.

[17] R.A. Serway, J.W. Jewett, *Physics for Scientists and Engineers*, Cengage Learning, 2008.

[18] G. Andersson, *Modelling and Analysis of Electric Power Systems*, lecture notes, ETH Zürich, 2008.

[19] W. van Westering et al., *Assessing and Mitigating the Impact of the Energy Demand in 2030 on the Dutch Regional Power Distribution Grid*, Proceeding of 2016 IEEE 13th International Conference on Networking, Sensing, and Control, Mexico City, april 28-30, 2016.

[20] M. van der Meulen, *Reconfiguring Electricity Distribution Networks to Minimize Power Loss*, master thesis, Radboud University Nijmegen, 2015.

[21] J.L. Kirtley Jr., *Introduction to Load Flow*, lecture notes for course *Introduction to Electric Power Systems*, Massachusetts Institute of Technology, 2011.

[22] P.C.M. Christianen, *Wisselstroom*, lecture notes for course *Elektriciteit en Magnetisme 1*, Radboud University Nijmegen, 2014.

[23] *Samen omschakelen, Jaarverslag 2016*, Alliander, 2017.