Karlsruhe Institute of Technology

A Block-Cut-Tree-based Switching Algorithm for Cacti

Bachelor Thesis of

Florian Krüger

At the Department of Informatics
Institute of Theoretical Computer Science

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Franziska Wegner, M.Sc.
	Matthias Wolf, M.Sc.

Time Period: 12th July 2018 – 11th November 2018

Statement of Authorship

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

Karlsruhe, 11th November 2018

Abstract

The Maximum Transmission Switching Flow Problem (MTSF) focuses on maximizing the power flow within an electrical power grid by allowing transmission lines to be removed from the grid, where the latter procedure is referred to as *switching*. MTSF is an *optimization* problem and known to be NP-complete, even on strongly restricted graph classes such as *cacti*. Therefore the standard approach is to solve an instance of its formulation as a Mixed Integer Linear Program (MILP).

In this thesis, we present an algorithm of exponential complexity to solve MTSF for *unbounded* cactus grids to optimality by utilizing the structure of its *Block-Cut-tree*. We achieve this by solving a power flow model for subgraphs and then composing partial solutions to an overall solution. We furthermore introduce a heuristic to gain improvements in terms of running time.
In conclusion we evaluate the results of our algorithm in comparison to solving the MILP on real power grid instances, that have been modified to represent cacti. We identify the cause for the exponential running time and also assess the solution deviation from optimality of the heuristic.

Deutsche Zusammenfassung

Bei dem Maximum Transmission Switching Flow Problem (MTSF) handelt es sich um ein *Optimierungsproblem*, das den Leistungsfluss innerhalb eines Stromnetzes maximiert. Um letzteres zu erzielen, gilt es eine Menge von Stromlinien zu ermitteln, die dabei aus dem Stromnetz entfernt werden. Es handelt sich hierbei um ein NP-vollständiges Problem, sogar wenn die Topologie eingeschränkt wird, wie etwa auf die Klasse der *Kaktusgraphen*. Demnach beruht der klassische Ansatz auf dem Lösen der Formulierung jenes Problems als Mixed Integer Linear Program (MILP).

In dieser Arbeit wird ein Algorithmus vorgestellt, der MTSF auf *unbeschränkten* Netzen in Form von Kaktusgraphen löst und dabei eine optimale Lösung produziert. Zur Hilfe gezogen wird dabei der *Block-Cut-tree* des Graphen, dessen Struktur zu Nutzen gemacht wird. Im Zuge dessen werde partielle Leistungsflüsse berechnet und weiterhin zu einer optimalen Gesamtlösung kombiniert werden. Desweiteren wird eine Heuristik vorgestellt, die eine Beschleunigung der Laufzeit des Algorithmus zur Folge hat.
Um die Qualität des Algorithmus bemessen zu können, dient das Lösen des MILP als Referenz. Dabei werden eine Reihe von Testinstanzen durchlaufen, die durch Modifizierung realer Stromnetzinstanzen zu Kaktusgraphen hervorgingen. Im Anschluss wird die Ursache für den exponentiellen Aufwand ermittelt und außerdem die Lösungsqualität der Heuristik bewertet.

Contents

1. Introduction

Fossil fuels and nuclear energy used to be the main providers globally for electricity in power grids and increasingly disappear as an option since the energy revolution. The demand for power, on the other hand, is still on the rise with renewable energies expected to fill this gap. Furthermore, the trend is towards the autonomous energy generation at the household level. These developments lead to a congestion of the power grid due to its topology remaining the same and an indefinite amount of concurrent energy generators.

A method to alleviate congestion is to temporarily remove transmission lines from the power grid, which is referred to as *switching*. This, on the other hand, is also a means of increasing the power flow and from this, the Maximum Transmission Switching Flow Problem (MTSF) arises. Its goal is to find the set of lines to be switched in order to maximize the generation of all energy producers. As it is an intractable problem, to date, no algorithm exists to solve MTSF in an efficient manner and so approximative solutions generated in a reasonable time are of use.

In this thesis, we are going to consider MTSF for *cacti*, a restricted class of graphs representing power grids. An approximation algorithm was already formulated by Grastien et al. [GRW$^+$18, p. 8], on which this thesis is based on. Their algorithm takes an *unbounded* power grid as input, which means that energy consumers do not have a lower demand. Here we will also use this type of model as our basis. But in contrast, we present an algorithm that solves MTSF to optimality with exponential complexity and subsequently, formulate a heuristic to gain improvements in terms of running time.

The thesis is structured as follows: In Chapter 2 we cover the foundations in graph theory and mixed integer linear programming, whereas Chapter 3 covers the related work in this field. In Chapter 4 we describe the basics of alternating current and its simplified version, as well as the flow model and the problem description for MTSF. Subsequently in Chapter 5 we present the algorithm and evaluate it on several test cases in Chapter 6.

2. Preliminaries

In this chapter, we will introduce formal basics that will be used throughout this work. As the topic of the thesis is all about modeling power grids, it is therefore necessary to mention the basic constructs and rules that are needed to achieve this goal. The main topics of interest are *Graph Theory* and *Mixed Integer Linear Programming*, starting out with the former.

2.1 Graph Theory

The most important part of graph theory forms the *graph* itself. It resembles a set of objects and connections, that exist between them. Those are denoted as *vertices* and *edges*.

Definition 2.1 (Simple Graph [Rah17, p. 11])**.** A simple **graph** $G = (V, E)$ is a 2-tuple, that consists of a finite set of *vertices* V and of a finite set of edges E. An edge $e \in E$ is a pair of vertices u and v, that connects both. u and v are then said to be *adjacent* and e is said to be *incident* to u and v. The number of incident edges of a vertex is referred to as its *degree*.

We denote the set of vertices of G by $V(G)$ and the set of edges by $E(G)$. Furthermore one can make the distinction, whether or not the *direction* of the edge is of importance. A *directed* edge is associated with a direction and is denoted by (u, v), where u is the initial and v the terminal vertex. An *undirected* edge is an edge directed in both ways and is denoted by $\{u, v\}$. This implies that v can be reached from u and u can be reached from v. In case of a directed edge, only the former is true.
If the edges of G are directed, then G is a *directed* graph, otherwise it is *undirected*. Note that an undirected edge can be replaced by two directed edges (u, v) and (v, u) and therefore, an undirected graph can always be transformed into a directed one. To draw a simple graph, each vertex is represented by a small circle and an undirected edge by a line connecting both of its vertices. In case of directed edges, we additionally draw an arrow at the end of the line, where it meets the terminal vertex, pointing towards it. If no prefix is given, the graphs considered in this thesis will be undirected.
Sometimes, a subset of edges of a graph is currently not of interest. Therefore, we introduce a compact notation. Let $G = (V, E)$ be a graph and $S \subseteq E$. Then we denote

$$G - S := (V, E \setminus S)$$

Next, the concept of *reaching* or making steps between vertices can be extended to result in a *path*: an arrangement of vertices, one has to traverse to get to one vertex from another.

Definition 2.2 (Path, Cycle [Rah17, pp. 31-32])**.** A **path** $p = (v_1, e_1, \ldots, e_{k-1}, v_k)$ in a given graph $G = (V, E)$ is an alternating sequence of vertices v_i and edges e_i of G, such that:

$$\forall e_i, v_i \in p : e_i = (v_i, v_{i+1}) \qquad 1 \leq i < k$$

Its length is defined by the number of its edges and is equal to $k-1$. Furthermore, a *simple* path can contain any vertex, except for the first and last vertex, at most once. If $v_1 = v_k$, then p is called a simple **cycle** and its length is equal to k.

In the following, only simple cycles are considered. For reasons of simplification the term simple is omitted and they are just referred to as cycles.

Definition 2.3 (Connected [Rah17, p. 32])**.** A graph is considered **connected**, if for every pair of vertices v_1 and v_2, there exists a simple path from v_1 to v_2. If a graph is not connected, it is called disconnected.

It is obvious, that for an undirected graph, a path from v_2 to v_1 always exists: it contains the same vertices, but in reverse order.

Definition 2.4 (Biconnected [Rah17, p. 39])**.** A graph is **biconnected**, if the resulting graph after removal of any vertex is still connected. A subgraph is denoted by maximal biconnected, if it contains as many vertices as possible, such that it is still biconnected. A *maximal* biconnected subgraph is called a biconnected component or *block*.

By *removing* a vertex, it is meant to delete the vertex from the set of vertices and also those edges incident to that vertex. As there can be several blocks in a graph, the vertices shared by more than one block are said to be the *cut vertices*. This can also be seen in Figure 2.1b.
Next, we will introduce a class of graphs with an interesting structure. This class is of special interest, as it is used to model a power grid later on.

Definition 2.5 (Cactus graph [Wes01, p. 160])**.** A **cactus graph** $G = (V, E)$ is a connected graph, in which every block is either an edge or a cycle.

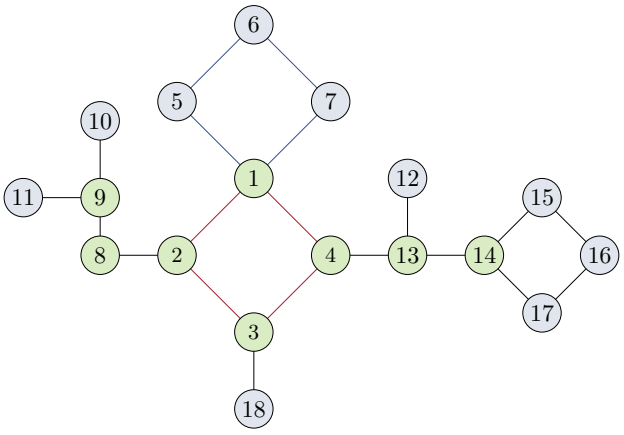
As the name implies, it resembles a cactus with leaves and spikes, when its drawn *planar*, that is no two edges are crossing. See Figure 2.1a for an example. Another important class of graphs for this work are *trees*. They are easy to *traverse*, that is to visit every of its vertex, and they will be later on used to store data, that is used during traversal.

Definition 2.6 (Tree [Rah17, p. 47])**.** A connected graph $G = (V, E)$ that does not contain cycles is called a **tree**. In a tree, there exists a path from any vertex u to any other vertex v with $u, v \in V$. Vertices with a degree of 1 are called *leaves*.

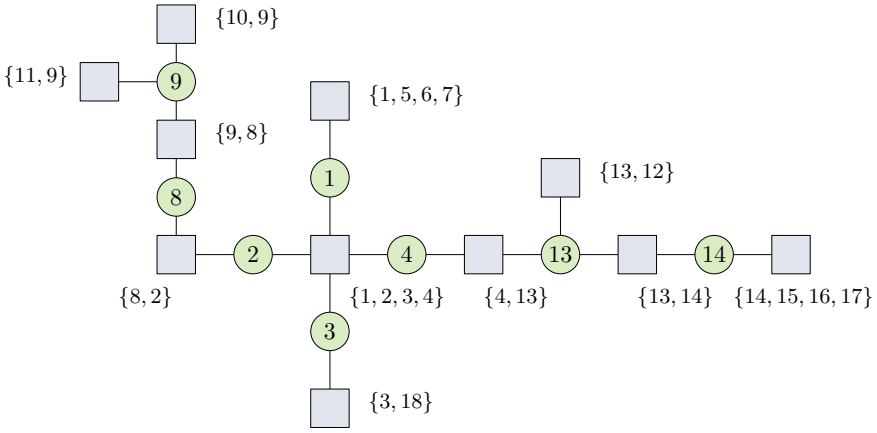
Trees in nature are not just placed arbitrarily, they all grow out of the ground, their *root*. One can apply this concept to graph theoretical trees by setting the root as one of its vertices. It is used for traversal as the starting point, so the order, in which vertices are visited, remains the same.

Definition 2.7 (Rooted tree [Rah17, p. 50])**.** A tree $\mathcal{R} = (V, E, v_R)$ with a vertex $v_R \in V$, declared as the *root* of the tree, is called a **rooted tree**. The vertices, that are not the root or a leaf, are called *inner vertices*.

When planar drawing a rooted tree, it is common to draw it bottom up unlike real world trees, e.g. the root is placed at the top and the leaves are placed at the bottom. An example for a tree is given in Figure 2.2.



(a) A cactus-graph with a cycle $(1, 5, 6, 7)$ highlighted in ■ and a biconnected component $(1, 2, 3, 4)$ highlighted in ■.



(b) The corresponding Block-Cut tree with vertices in ■ representing blocks and vertices in ■ representing cut vertices.

Figure 2.1

With a given rooted tree, one would like to perform actions on its vertices, that are not independent of each other. Therefore, an order in which vertices are processed has to be declared at first. This is referred to as tree *traversal*, one of them being *reverse level-order*:

Algorithm 2.1: Reverse Level-Order Traversal

Input: Rooted tree $\mathcal{R} = (V, E, v_R)$, Stack S, Queue Q
Output: Queue Q of vertices $\in V(\mathcal{R})$

```
1 Q.ENQUEUE(v_R)
2 while Q NOT EMPTY do
3     current ← Q.DEQUEUE
4     S.PUSH(current)
5     forall v ∈ current.CHILDREN do
6         Q.ENQUEUE(v)
7 while S NOT EMPTY do
8     Q.ENQUEUE(S.POP())
9 return Q
```

The set CHILDREN of a vertex $v \in V(\mathcal{R})$ stores all of its j succeeding vertices w_i, $i \in \{1, \ldots, j\}$, which, in this context, are all vertices adjacent to v. After they were inserted,

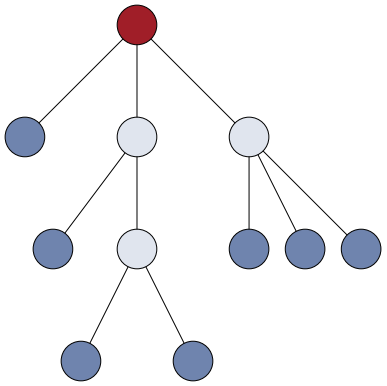


Figure 2.2: A rooted-tree with root ■, inner vertices ■ and leaves ■.

we remove the adjacency of all pairs v and w_i. Note that this set is only formed during each iteration of the WHILE-loop in Line 2 and not as an initialization routine. And since the iteration starts at the root vertex v_R, we get the desired result.
S and Q are *data structures* and represent a *stack* and *queue* respectively. They allow for elements to be stored and retrieved again. The methods PUSH and POP are used to place/retrieve an element at/from the top of the stack. Similarly the methods ENQUEUE and DEQUEUE are used to place/retrieve an element at/from the end/beginning of the queue. This can easily be comprehended if one imagines a stack of papers on a desk or a queue of people at a market.

A cactus graph can be transformed into a special tree, without loss of information. That is, the original graph could be reconstructed from that tree. Later in this thesis, we want to examine individual blocks and their contribution to power. This special tree used is the *Block-Cut-tree*.

Definition 2.8 (Block-Cut-tree [Wes01, p. 156])**.** The set of blocks of G is denoted by B and the set of cut vertices as C. A **Block-Cut-tree** $\mathcal{BC} = (B, C)$, shortened BC-tree, of G is a tree of which a vertex v either represents a block $b \in B$ or a cut vertex $c \in C$. There exists an edge between b and c, if $c \in V_B$ (b contains v).

This implies that an edge does not exist between two blocks or two cut vertices. However, a cut vertex can be adjacent to several blocks, since it can separate more than one block. An example for a Block-Cut tree can be seen in Figure 2.1b. For the sake of brevity, blocks are only labeled with their set of vertices.

2.2 Mixed Integer Linear Programming

Mixed Integer Linear Programming (MILP) [NW99] is the process of *maximizing* an *object function*, that has to be *linear*. It is used to solve *optimization* problems, where the *best* solution amongst all possible/*feasible* solutions is to be determined and is therefore categorized under linear optimization.
The first step is to create the actual *Mixed Integer Linear Program*. It consists of the object function $\mathbf{z}$, that has to satisfy certain constraints. The program is written as

$$max\{\mathbf{z} := \vec{c}^{\top}\vec{x} + \vec{h}^{\top}\vec{y}\} \tag{2.1}$$

with

$$\vec{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{Z}_+^n, \quad \vec{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_p \end{bmatrix} \in \mathbb{R}_+^p, \quad \vec{c} = \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix} \in \mathbb{R}^n, \quad \vec{h} = \begin{bmatrix} h_1 \\ \vdots \\ h_p \end{bmatrix} \in \mathbb{R}^p$$

$\vec{x}$ and $\vec{y}$ are the unknown variables, that are to be found. The term *integer* derives from the vector $\vec{x}$ being a member of the integer vector space. And the additional term *mixed* arises, because $\vec{y}$ is from a different vector space. $\vec{c}$ and $\vec{h}$ are coefficients of the unknown variables and can be seen as *costs*.
The constraints, **z** has to suffice, are of form

$$\mathbf{A}\vec{x} + \mathbf{G}\vec{y} \leq \vec{b} \tag{2.2}$$

with

$$\mathbf{A} = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix} \in \mathbb{R}^{m \times n}, \quad \mathbf{G} = \begin{bmatrix} g_{11} & \dots & g_{1p} \\ \vdots & \ddots & \vdots \\ g_{m1} & \dots & g_{mp} \end{bmatrix} \in \mathbb{R}^{m \times p}, \quad \vec{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix} \in \mathbb{R}^m$$

The matrices **A** and **G** denote the coefficients for the integer and real variables respectively. $\vec{b}$ is simply called the *right-hand side*.
Further, the *feasible region* S is defined as the set of tuples, that satisfy Equation (2.2). More specific

$$S = \{\vec{x} \in \mathbb{Z}_+^n, \vec{y} \in \mathbb{R}_+^n : \mathbf{A}\vec{x} + \mathbf{G}\vec{y} \leq b\}$$

An element of S is then considered as *feasible solution*. Amongst those, the *optimal solution* is to be found and it is defined as

$$(\vec{x^0}, \vec{y^0}) \in S : \vec{c}^\top \vec{x^0} + \vec{h}^\top \vec{y^0} \geq \vec{c}^\top \vec{x} + \vec{h}^\top \vec{y} \quad \forall (\vec{x}, \vec{y}) \in S$$

$\vec{c}^\top \vec{x^0} + \vec{h}^\top \vec{y^0}$ is then denoted as the *optimal value* of the solution.

The drawback of MILP is that it is an *NP-complete* problem and to date, there are no algorithms that produce optimal solutions with polynomial complexity. However the methods and heuristics used by modern MILP solvers produce sufficient solutions in a reasonable amount of time.

3. Related Work

The situation of congested resources in a network can be found in several domains, one of them being road networks with car traffic. Dietrich Braess has examined this very issue and formulated the eponymous *Braess' Paradox* [Bra68]. It states that given a fixed number of users in a congested road network, adding a road to that network can cause a redistribution of traffic flow, which can lead to an overall increase of traveling times for each participant. This counterintuitive behavior occurs because "users attempt to minimize their own travel time while ignoring the effect of their decisions on other travelers" [PP97, p. 2]. It implies in turn that *removing* a road from the network can improve its overall traffic flow.

Witthaut and Timme [WT12, p. 3] have shown, that the paradox also appears in *oscillator networks*, which can be used to model *dynamic power grids* on a coarse scale. An example for this kind of network is a network of rotating machines e.g. water turbines. They have shown that in these networks, a phase-locked state between energy producers and consumers can potentially be decreased or even destroyed, when adding a new link. Here phase-locked means that the phase difference for any two machines is fixed and such a state is considered stable. Otherwise, transmission of electrical power is inhibited and can even lead to a "global power outage" [WT12, p. 10].
Another manifestation can even be seen in the field of quantum physics as Pala et al. [PBL$^+$12] have shown, considering semiconductors on a *mesoscopic* level with a magnitude of 10^{-6} meters. They examined a network of two conducting paths connecting electron source and drain. Upon adding a third path to that network, they intuitively expected an increased total current. But instead they observed a decrease in current, even when increasing its width.

As indicated in the first paragraph, Braess' paradox can be exploited and put to use. This can be seen in the context of power grids, where *transmission lines* represent connections in order for electrical power to be transmitted. A removal of such a line is referred to as *switching*, which at first has been a practice to reduce line overloads [KM80]. But further studies have shown that it also brings along other positive effects.

Fisher et al. [FOF08] discovered *operational costs*, i.e. costs of operating power generators, to be decreased by 25% at most when performing switching on the 118-bus IEEE test case, a simple approximation of the American electric power system. They declared the problem of finding a subset of transmission lines to be switched, such that overall generator costs are minimized, as the Optimal Transmission Switching Problem (OTS). Its formulation is based on the *direct current* (DC) model, which is a linearization of the

alternating current (AC) model. The former model is a result of the latter model after performing some simplifications which are mentioned later on in this thesis. Although the DC model is not as accurate for power grids as the AC model, it still yields acceptable results and can be solved in a reasonable time via techniques of linear optimization [OCS04, p. 2].
Lehmann et al. [LGH14] have proven OTS to be *NP-complete*. To date, there does not exist an algorithm with a polynomial running time to solve any problem that is NP-complete. Therefore in case of an optimization problem, simpler methods with a running time less than *exhaustive search* that still produce solutions near the optimum are required, such as *heuristics* or *approximation algorithms*.
Barrows et al.[BB11] further analyzed OTS on the RTS-96 network, which consists of three loosely-connected 24-bus power systems. They noticed increased operational savings when switching a small number of lines during peak periods of the grid, that is, periods of highest utilization. Their results also hint that effects of OTS are relatively localized and that the network could be decomposed into smaller networks, on which the switching is then applied.
Since large-scale networks add many variables to the OTS problem, Barrows et al.[BBB13] made an approach to formulate a heuristic that only considers specific candidates for switching, in order to reduce the overall search space. Their idea was to pick possible switching candidates during a pre-screening according to their *Available Branch Capacity* (ABC), a measure of spare capacity for each transmission line. They limited the set of switchable lines to those that cause an increase of ABC on congested lines, that is, lines with no spare capacity. From that set, they switched a fixed size of lines causing the search space to be reduced even further. The heuristic was then also validated on the RTS-96 network for a load case of 24 hours. During an hour of this case, they were able to compute optimal cost saving within 3.6 minutes, whereas non-screened OTS took 4.3 hours.

Besides savings in operational costs, another observed benefit of switching is the possibility to increase the *power flow* within the power grid. This is due to power flow "depending on potentials at the end of nodes of transmission lines" and this "creates additional (cyclic) constraints on the flow that can be eliminated by switching" [LGH15, p. 2]. As a result, Lehmann et al. [LGH15] formulated the Maximum Power Flow Problem and the Maximum Transmission Switching Flow. The objective of the former is to maximize the power flow accumulated over every generator in the power grid, whereas the latter denotes the maximum power flow when allowing transmission lines to be switched. Furthermore they have proven MTSF to be NP-complete on *planar* graphs, i.e. graphs that can be embedded in the plane without edge crossings, and on *cactus* graphs.
Grastien et al. [GRW$^+$18] showed that MTSF is also NP-complete, even if the network only contains one generator and one consumer. As a result they formulated an algorithm for MTSF on such a network, that is further restricted. They also have given an approximation algorithm for MTSF on cacti with a factor of 2, meaning the quality of the computed solution is guaranteed to be at least as good as 50% of the optimal solution. Grastien et al. achieve this by switching the line with the smallest capacity in every cycle of the network. The overall running time of their algorithm is a function linear in the total number of nodes of the power grid.

4. Model

In this chapter, we introduce the utilized power flow models, denoted by *AC* and *DC* model, where the latter is the linearization of the former. Afterwards we present the formulation of the *Maximum Transmission Switching Flow Problem*, which is the main focus of this thesis.

4.1 Alternating Current

Alternating Current (AC) is nowadays the common form to provide electricity across a power grid and before modeling a power grid, it is necessary to enumerate the fundamentals to do so. The term *alternating* originates from the fact that the voltage has a periodic, sinusoid waveform.

Definition 4.1 (Electric circuit [Bau13, p. 46])**.** An *electric circuit* consists of a *power source* and a *load*. They are connected through *conductors* which allow for electrical power to flow from source to load. A power source *delivers* and a load *absorbs* electric power.

Definition 4.2 (AC voltage [Bau13, p. 279])**.** The electrical *voltage* $\underline{\mathrm{u}}(t)$ of an AC circuit is described by

$$\underline{\mathrm{u}}(t) = U_0 \cdot \sin(\omega t + \phi),$$

where

$t \in \mathbb{R}_{\geq 0}$	represents a point in time,
$\omega := 2\pi \cdot f$	denotes the *angular frequency* with frequency f in Hertz [Hz],
ϕ	is the *phase angle* in radians and
U_0	is the *amplitude* of the waveform and represents the maximum voltage.

The phase angle indicates the shift of voltage *relative* to the base signal $\sin(\omega t)$. Applying an AC voltage to an electrical component causes an electrical *alternating current* $\underline{\mathrm{i}}$, which is shifted to the voltage by a varying amount, depending on which component is present. These components are either a *resistor*, *capacitor* or *inductor*.

Definition 4.3 (Impedance [Sch18, p. 76-77])**.** Electrical *impedance* $\underline{\mathrm{Z}}$ of an AC circuit is the ratio between voltage $\underline{\mathrm{u}}$ and current $\underline{\mathrm{i}}$. It is given by the complex notation

$$\underline{Z} = \frac{\underline{\mathrm{u}}}{\underline{\mathrm{i}}} = R + j \cdot X,$$

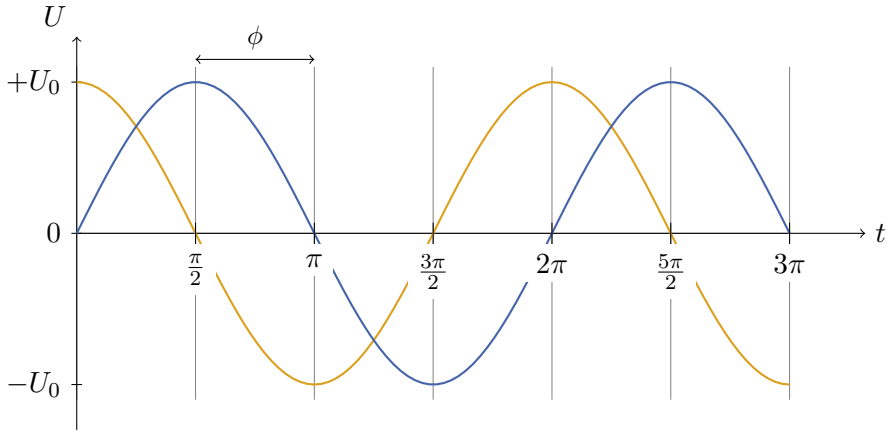


Figure 4.1: AC voltage $\sin(t)$ with $f = \frac{1}{2\pi}, \omega = 1$ and amplitude U_0. $\sin(t + \frac{\pi}{2})$ in , it is the same signal, but shifted by a phase angle ϕ of $\frac{\pi}{2}$.

where R is the electrical ohmic resistance and X is the electrical reactance, the ratio of voltage and current for a given AC component.

Definition 4.4 (Admittance [Sch18, p. 76])**.** Electrical *admittance* is defined as the reciprocal of the impedance, that is

$$\underline{Y} = \frac{1}{\underline{Z}} = \frac{R}{R^2 + X^2} + j \cdot \frac{-X}{R^2 + X^2} = G + j \cdot B,$$

where R is the electrical resistance and X is the electrical reactance.

Admittance describes the ease at which an AC circuit allows a flow of current in contrast to impedance. G and B are referred to as *conductance* and *susceptance*. Both are of importance when it comes to modeling power grids, as they are used to characterize transmission lines. In this context they are referred to as *line parameters*.

Definition 4.5 (AC Power [Sch18, pp. 155-156])**.** The electrical *power* S in an AC circuit is given by the complex notation

$$\underline{S} = P + j \cdot Q$$

P is the *real* power, Q the reactive power and j is the imaginary unit. The *apparent* power refers to the absolute value of the power $|\underline{S}|$.

4.2 Power Grid Models

Now we will take a closer look at power grids and its component units. As a reference, I will be using the *IEEE 14-bus test case*, Figure 4.2, provided by the University of Washington [Uni99].
The power grid consists of the following:

- *branches*, which represent the *transmission lines*
- *buses*, which connect multiple branches, *generators* and/or *loads*
  - *generator bus*: a bus, that is connected to a power supplier, a *generator*
  - *load bus*: a bus with a demand for power
- *transformers*, which transform voltage into a higher or lower one

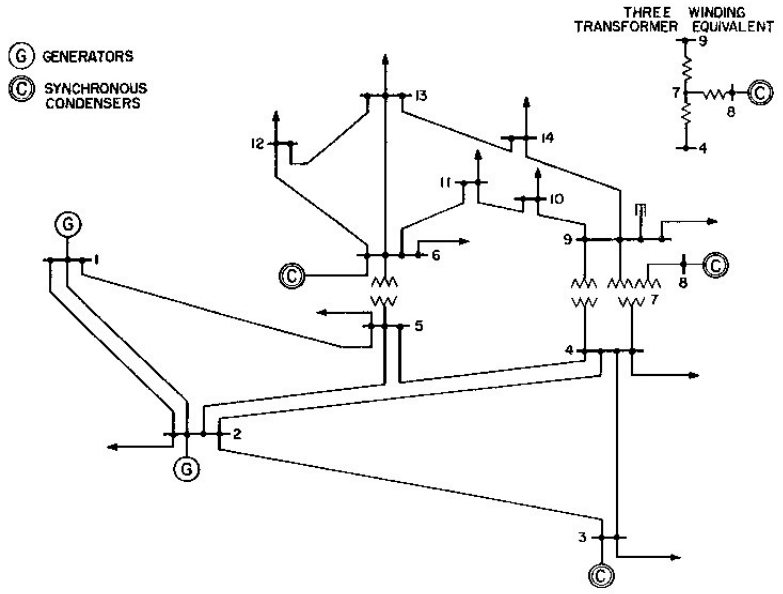


Figure 4.2: Diagram of the IEEE 14-bus test case [Uni93]. The buses are the numbered, horizontal bars and the lines between them are the branches. Note that there can be more than one branch connecting two buses. The generators are the buses labeled with a G and C respectively, the latter are condensers. And loads are the buses, that have an outward pointing arrow attached to them.

4.2.1 DC Model

For an AC model, the set of equations to be solved, in order to perform the power flow study, are of non-linear form. They are hard to solve and also require many parameters. Therefore, we introduce simplifications to the AC model. The outcome is denoted by the *DC model*. The set of equations it yields are linear and thereby solvable via mixed-integer-programs, which in turn can be computed by the method described in Section 2.2.

To transform the AC model into the DC model, four simplifications [ZMST11] are made:

- only the real power $P = \mathrm{Re}(S)$ is considered
- branches can be considered lossless, so the transmission line parameters branch resistances r_s and charging capacitances b_c are negligible; the series reactance is denoted by x_s
$$y_s = \frac{1}{r_s + j \cdot x_s} \approx \frac{1}{j \cdot x_s} \qquad r_s, b_c \approx 0$$
- the differences of voltage angles Θ across branches are small enough to satisfy
$$\sin(\Theta_f - \Theta_t - \Theta_{\text{shift}}) \approx \Theta_f - \Theta_t - \Theta_{\text{shift}}$$
where Θ_f and Θ_t is the voltage angle of the *from* and *to* bus of a branch respectively
- all bus voltage magnitudes are close to 1 *per-unit*
$$v_i \approx e^{j\Theta_i}$$

In the *per-unit* system (p.u.) [GSO12, pp.108-109], power system quantities such as voltage, current, power and impedance are given by *per-unit quantities*. A per-unit quantity is given by

$$\text{per-unit quantiy} = \frac{\text{actual quantity}}{\text{base value of quantity}}$$

where *actual quantity* is the value of the quantity in the actual units. The base value has the same units, making the per-unit quantity dimensionless. It is sufficient to pick two base values arbitrarily, which are usually the base voltage V_{base} and base power S_{base}. From there, one can derive the other base values as follows

$$I_{\text{base}} = \frac{S_{\text{base}}}{V_{\text{base}}},$$
$$Z_{\text{base}} = \frac{V_{\text{base}}}{I_{\text{base}}}.$$

Considering a network of n_b buses, n_l branches and n_g generators, the power flows at the *from* ends of a branch are now given by

$$\vec{\mathrm{P}}_{\mathrm{f}}(\vec{\Theta}) = \mathbf{B}_f \cdot \vec{\Theta} + \vec{P}_{f,\text{shift}}. \tag{4.1}$$

$\vec{\Theta}$ denotes the $n_b \times 1$ vector of voltage angles of each bus. $\vec{P}_{f,\text{shift}}$ is a vector of dimension $n_l \times 1$ and its i^{th} element is defined as $-\Theta^i_{\text{shift}} \cdot b_i$ with $b_i = \frac{1}{x^i_s \cdot \tau_i}$ and τ_i being the *tap ratio*, the ratio of turns of primary and secondary coil in a transformer. Then, each b_i is store in a $n_l \times 1$ vector $\vec{B}_{\text{ff}}$.
The matrix $\mathbf{B}_f$ is defined as $\mathbf{B}_f = [\vec{B}_{\text{ff}}] \cdot (\mathbf{C}_f - \mathbf{C}_t)$, where $[\vec{B}_{\text{ff}}]$ denotes the quadratic matrix, whose $(i,i)^{th}$ element is b_i and 0 otherwise. Connection matrices are given by $\mathbf{C}_f$ and $\mathbf{C}_t$, where the $(i,j)^{th}$ element of $\mathbf{C}_f$ and the $(i,k)^{th}$ element of $\mathbf{C}_t$ are 1, if branch i connects from bus j to bus k and 0 otherwise [ZMST11, pp. 16–22]. Due to the assumption of lossless branches, the branch flows at the *to* ends are given by

$$\vec{\mathrm{P}}_{\mathrm{t}} = -\vec{\mathrm{P}}_{\mathrm{f}}$$

and thereby, the direction only defines the sign.

Any electrical circuit obeys *Kirchhoff's circuit laws*. The variation for the DC model are stated in the following two definitions:

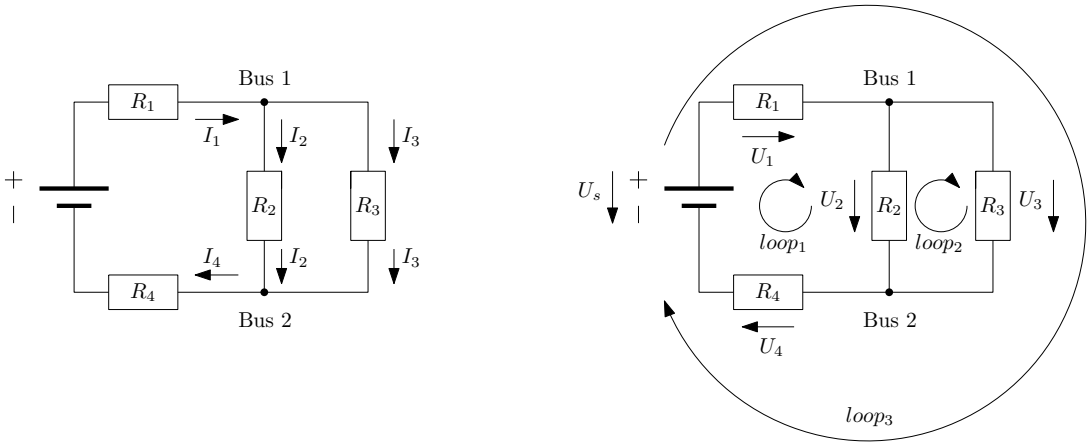
Definition 4.6 (Kirchhoff's current law)**.** *Kirchhoff's current law*, **KCL** in short, states that at any junction in an electric circuit, the sum of all incoming currents is equal to zero. A junction refers to a point in which multiple conductors merge. Let n be the number of incident conductors to a given junction and I_j the electric current flowing on conductor of index j. The KCL can then be expressed as

$$\sum_j I_j = 0 \qquad j \in \{1, \ldots, n\}$$

Definition 4.7 (Kirchhoff's voltage law)**.** *Kirchhoff's voltage law*, **KVL** in short, states that the sum of all voltages in any loop of an electric circuit is equal to zero. A loop is a cycle in an electric circuit. Let m be the number of voltages contained in a given loop and U_j one of these voltages. The KVL can then be expressed as

$$\sum_j U_j = 0 \qquad j \in \{1, \ldots, m\}$$

These laws have to be considered in this work because a power grid in form of a cactus graph can exhibit loops and multiple transmission lines can merge into each other. For an illustration of both laws, see Figure 4.3.



(a) Kirchhoff's current law: the sum of currents at a junction is equal to 0. For Bus 1, it has to hold: $I_1 = I_2 + I_3$ and for Bus 2: $I_2 + I_3 = I_4$.

(b) Kirchhoff's voltage law: the sum of voltages in a closed loop is equal to 0. All possible loops are drawn into the circuit. For $loop_1$, it has to hold: $U_1 + U_2 + U_4 = U_s$ and for $loop_2$: $U_3 = U_2$.

Figure 4.3: Kirchhoff's circuit laws illustrated.

4.2.2 Graph Representation

The representation of a power grid is straightforward, with the simplification that there exists only one branch between two buses.
Buses are represented by *vertices* and branches by *edges*. Generator and load buses induce the set of *generators* and *loads*. Note that a vertex can be generator and load simultaneously in this context. The graph representation of the 14-bus power grid, Figure 4.2, can be seen in Figure 4.4.

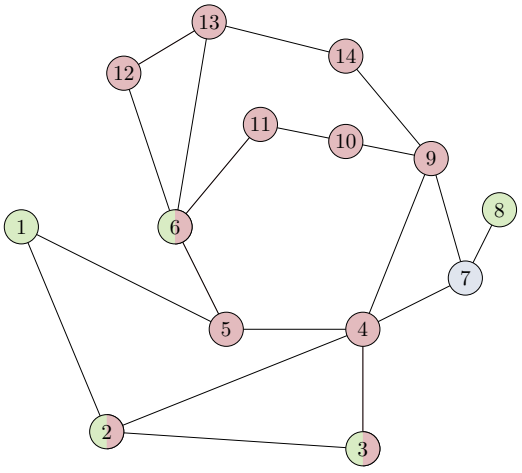


Figure 4.4: 14-bus power grid as a graph. vertices are loads and are generators. Buses 2,3 and 6 are both generators and loads, whereas bus 7 is a transformer and none of the prior.

4.3 Model Formulation

Let $G = (V, E)$ be a graph with vertices V and edges E. The subset $V_G \subseteq V$ represents the *generators* and the subset $V_C \subseteq V$ the *producers*. Unlike Figure 4.4, we set producers to not be consumers also and vice versa, implying $V_G \cap V_C = \emptyset$.

Moreover, transmission lines cannot transfer an infinite amount of power, as its conductors have a thermal limit. Surpassing those limits would lead to instability of the line, up to even bursting. To model this constraint, a *capacity* function $cap\colon E \to \mathbb{R}_{\geq 0}$ will be used. We then also take into consideration the transmission line parameters *electical reactance* $x\colon E \to \mathbb{R}_{\geq 0}$ and its reciprocal, the *susceptance* $b\colon E \to \mathbb{R}_{\geq 0}$ [GRW+18, pp. 3–4].

With these parameters, the power grid can be described by the 5-tuple $\mathcal{N} = (G, V_G, V_C, cap, b)$, and $\mathcal{N}$ is now denoted as the *network*. Furthermore, for every generator and consumer in $\mathcal{N}$, we set it to be a degree 1 vertex. This means we replace the original vertex by an edge with the generator/consumer located at the end of it. The capacity of it is set to be the maximum generation/demand.
The next step is to look at how the transmission of electrical power can be modeled. For this purpose, the concept of a *flow* is introduced. A flow is a function f that assigns a real number to every edge in the network: $f\colon E \to \mathbb{R}$. One can think of flow as the transport of units from a *source* to a *sink* like in a water supply system.
The net flow of a vertex $u \in V(G)$, that is, the in- and outgoing flow of this vertex, is given by $f_{\text{net}} := \sum_{\{u,v\}\in E(G)} f(u,v)$. For f to be considered a flow, it has to satisfy four constraints:

$$f(u,v) = -f(v,u) \qquad \forall (u,v) \in E(G), \tag{4.2}$$
$$f_{\text{net}}(u) = 0 \qquad \forall u \in V \setminus (V_G \cup V_C), \tag{4.3}$$
$$-\infty \leq f_{\text{net}}(u) \leq 0 \qquad \forall u \in V_C, \tag{4.4}$$
$$0 \leq f_{\text{net}}(u) \leq \infty \qquad \forall u \in V_G. \tag{4.5}$$

Equation (4.2), or the *skew-symmetry* property, states that the direction of a flow is dictated by its sign. Equation (4.3) states that every unit flowing into a vertex must also flow out of it, except for generators and consumers. For the latter two, Equation (4.4) and Equation (4.5) applies, where the former states that consumers must receive units in a range of zero to negative infinity. The negative sign here implies an incoming flow. And similarly, generators produce units in a range of zero to infinity. This is the same as the previous constraint, but multiplied by a factor of -1. And since the range of production and consumption of flow respectively, begins at zero, we refer to $\mathcal{N}$ as an *unbounded* network. Furthermore a flow is considered to be *feasible*, if it does not surpass the thermal limits of a line, that is, it holds

$$|f(u,v)| \leq cap(u,v) \qquad \forall (u,v) \in E. \tag{4.6}$$

For a feasible flow, we can then define its *value* $F(\mathcal{N}, f) = \sum_{u\in V_G} f_{\text{net}}(u)$ as the accumulated generation for each generator. A feasible flow f on $\mathcal{N}$ maximizing $F(\mathcal{N}, f)$ is called Maximum Flow (MF) and its value is denoted by $\text{OPT}_{\text{MF}}(\mathcal{N}) = \max F(\mathcal{N}, f)$.

4.4 The Maximum Transmission Switching Problem

A feasible flow does not take into account physical constraints. These are set by Kirchhoff's voltage law and have to be included into the previous definition, whereas Kirchhoff's current law is already covered by Equation (4.3).

Electrical voltage is defined as the difference of an *electrical potential* of two points u and v. The electrical potential for an AC circuit is given by the *phase angle* $\Theta\colon V \to \mathbb{R}$. Phase angles are bounded by a minimum and maximum value $\Theta_{\min}$ and $\Theta_{\max}$ respectively. Therefore, for a network it has to hold

$$\Theta_{\min}(u) \leq \Theta(u) \leq \Theta_{\max}(u) \qquad \forall u \in V(G), \tag{4.7}$$
$$b(u,v) \cdot (\Theta(u) - \Theta(v)) = f(u,v) \qquad \forall (u,v) \in E(G). \tag{4.8}$$

Equation (4.8) represents Equation (4.1) of the DC approximation, but here we set $\vec{\Theta}_{\text{shift}}$ to be 0.
For an arbitrary vertex, the phase angle is set to 0 and that vertex is then referred to as the *slack* vertex. All other phase angles are then determined by using the slack as a reference. A flow that obeys Equation (4.7) and Equation (4.8) is considered *electrically feasible*. An electrically feasible flow f maximizing $F(\mathcal{N}, f)$ is called a Maximum Power Flow (MPF) and is denoted by $\text{OPT}_{\text{MPF}}(\mathcal{N}) = \max F(\mathcal{N}, f)$.

Transmission lines of $\mathcal{N}$ can be either active or inactive, or to put it in technical terms: *switched on* or *switched off*. This is represented by the function $z\colon E(G) \to \{0, 1\}$:

$$z(u,v) = \begin{cases} 1, & \text{if the line is switched on and} \\ 0, & \text{otherwise.} \end{cases}$$

The transmission lines in switched off state form the set of *switched* edges or *switching* $S := \{e \in E(G) \mid z(e) = 0\}$ of the network $\mathcal{N}$, thereby inducing a second network $\mathcal{N} - S$. We have to consider these edges within the KVL, therefore Equations (4.6) and (4.8) now become Equations (4.9) and (4.10)

$$|f(u,v)| \leq z(u,v) \cdot cap(u,v) \qquad \forall (u,v) \in E(G) \qquad (4.9)$$

$$b(u,v) \cdot z(u,v) \cdot (\Theta(u) - \Theta(v)) = f(u,v) \qquad \forall (u,v) \in E(G) \qquad (4.10)$$

With this, another type of flow can now be introduced: the Maximum Transmission Switching Flow (MTSF). Its task is to maximize a power flow while allowing edges to be switched off. This can be capsuled in a more precise formulation:

Maximum Transmission Switching Flow Problem

Instance: A network $\mathcal{N} = (G, V_G, V_C, cap, b)$.

Objective: Find a switching $S \subseteq E(G)$, such that $\text{OPT}_{\text{MPF}}(\mathcal{N} - S)$ is maximal.

Its value is given by $\text{OPT}_{\text{MTSF}}(\mathcal{N}) := \max_{S \subseteq E(G)} \text{OPT}_{\text{MPF}}(\mathcal{N} - S)$. Before discussing the benefits of maximum transmission switching, let us first emphasize the difference between a maximum flow and a maximum power flow by taking a look a the following network:

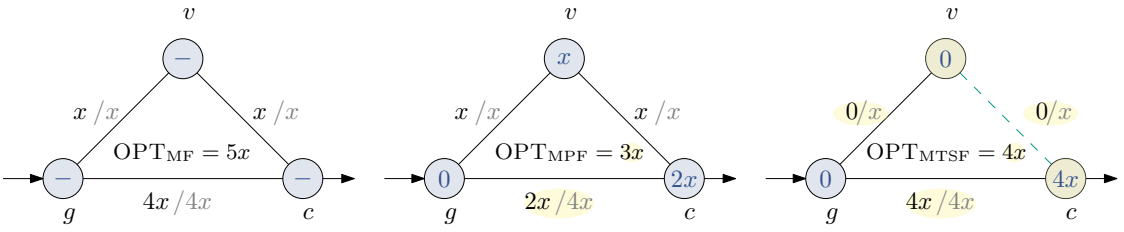
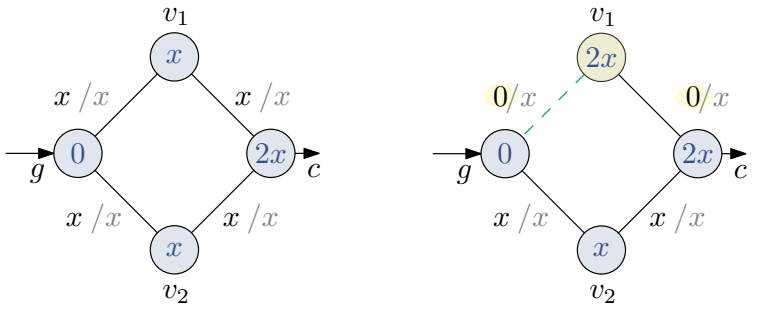


Figure 4.5: A network $\mathcal{N}$ for which the MF, MPF and MTSF have been computed [GRW+18, p. 3]. It consists of three vertices of which g and c are generator and consumer respectively. The transmission lines have susceptances of 1 and are labeled with capacities ■, vertices are labeled with phase angles ■. Successive differences of parameters are highlighted in ■.

In Figure 4.5a the maximum flow is computed and it yields a value of $5x$, whereas in Figure 4.5b the maximum power flow yields a smaller value of $3x$ due to the KVL. Line $\{g, c\}$ cannot be congested because the voltage differences for $\{g, v\}$ and $\{v, c\}$ would imply

a flow of $2x$ and therefore interfere with the capacity constraint (see Equation (4.6)).
Now if we allow for lines to be switched off, we can achieve a greater result as Figure 4.5c shows. By switching line $\{v, c\}$, the value now increases to $4x$. This is due to $\mathcal{N}$ not containing a cycle anymore and the voltage angles now being less constrained. Note the phase angle of v now is 0 and not x for example, because otherwise we would violate the flow conservation constraint (see Equation (4.3)), as v would have to pass on a flow of x, which is not possible due to $\{v, c\}$ being switched off. Also note that $\text{MTSF}(\mathcal{N} - \{v, g\})$ does not reach the upper limit of the maximum flow.
This example demonstrates that allowing for transmission lines to be switched off can increase the overall maximum power flow. However, this is not always the case as we can observe in the following example:



$\text{OPT}_{\text{MPF}}(\mathcal{N}) = 2x$ (a)

$\text{OPT}_{\text{MPF}}(\mathcal{N} - \{g, v_1\}) = x$ (b)

Figure 4.6: A network $\mathcal{N}$ that consists of a cycle of symmetric paths, susceptances are set to be 1 [GRW+18, p. 23]. It consists of one generator g, one consumer c and vertices v_1, v_2 in between, for which holds: $\text{OPT}_{\text{MPF}}(\mathcal{N} - \{g, v_1\}) = x = \frac{1}{2}\text{OPT}_{\text{MPF}}(\mathcal{N})$.

The edges of the network in Figure 4.6a all have equal parameters and there are two edge-disjoint paths from g to c, both of which have the same number of edges. Therefore they are referred to as *symmetric* paths. Upon switching an edge in such a network, the value of the resulting MPF decreases. Grastien et al. [GRW+18, p. 9] showed that an electrically feasible flow exists when switching the edge with the smallest capacity in a cactus network $\mathcal{N}$, and that its value is equal to $\frac{1}{2}\text{OPT}_{\text{MF}}(\mathcal{N})$.

5. Algorithm

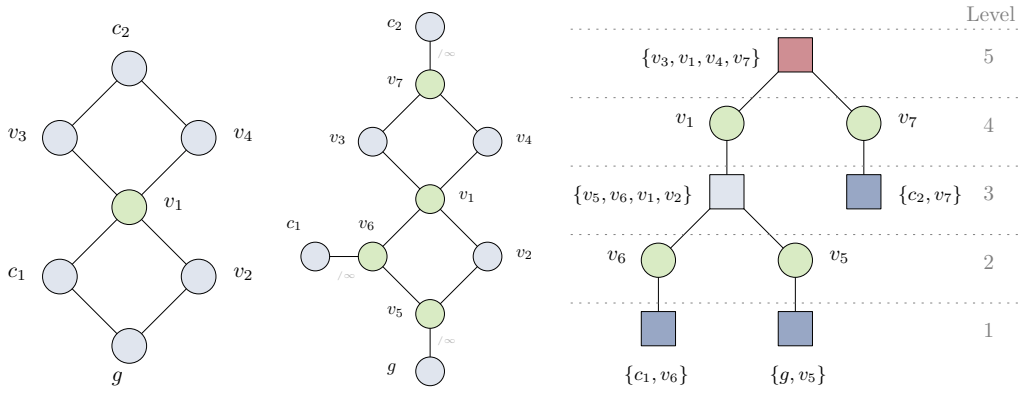
In this chapter, an algorithm based on the observations of the previous chapters, as well as the idea behind its procedures, is presented. We then mention its drawbacks and formulate a way to improve it in terms of running time.

5.1 Exponential Time Algorithm for MTSF on Cacti

The algorithm takes as input the network $\mathcal{N} = (G, V_G, V_C, \text{cap}, b)$ and operates on the BC-tree $\mathcal{BC} = (B, C)$ of graph $G(\mathcal{N}) = (V, E)$. Since we set generators and consumers to be vertices of degree 1, we can make two observations:

1. Each generator/consumer resides in its own block:
   $\forall v_i, v_j \in (V_G \cup V_C), v_i \neq v_j, \beta_m, \beta_n \in B(\mathcal{BC})\colon v_i \in V(\beta_m), v_j \in V(\beta_n) \implies \beta_m \neq \beta_n$
2. A block that contains a generator/consumer is a leaf of $\mathcal{BC}$.

With Observation 1 and because a network contains at least one generator and one consumer, both of which are set to be degree 1 vertices, a block with degree greater or equal to 2 always exists and can be set as the root. This step is visualized in Figure 5.1:



(a) Original network with generator g and consumers c_1 and c_2. Cut vertices are colored in .

(b) Transformation into degree 1 vertices.

(c) Reverse level-order traversal of $\mathcal{BC}$ which always starts out at the lowest level. Squares represent blocks, leaves are and the root is .

Figure 5.1

Once a root is selected, we refer to $\mathcal{BC}$ as the rooted BC-tree $\mathcal{R}$. The next step is to define the order in which the vertices of $\mathcal{R}$ are visited because we want to collect the contributions of power flow from every block. We also want to visit the leaves first, as generators and consumers only reside in these blocks. Therefore we *traverse* $\mathcal{R}$ in *reverse level-order* (see Algorithm 2.1) and an illustration can be seen in Figure 5.1c.
Furthermore, $\mathcal{R}$ holds two vertex based functions PARENT: $(B \cup C) \to (B \cup C)$ and CHILDREN: $(B \cup C) \to \mathcal{P}(B \cup C)$. The former represents a pointer to the parent vertex and the latter a pointer to a set of child vertices, respectively.

Let $\beta := (V, E, V_{cv})$ be a block with $V \subseteq V(G)$ being the set of its vertices, $E \subseteq E(G)$ being the set of its edges and $V_{cv} := \{c \mid c \in C \text{ and } c \text{ is adjacent to } \beta\}$ being the cut vertices, that belong to this block. To compute an MTSF of $\mathcal{N}$, we have to consider all switchings in each block. For each switching we then compute an MPF and collect the power generation in the subtree $\mathcal{G}_{\text{sub}}$ of that block, the power flow into the parent cut vertex, which we refer to as the *port*, as well as the currently considered switched edge and store them together in a 3-tuple that we denote as a *label* l:

$$\mathrm{l} := (\mathrm{gen}, \mathrm{f}, \mathrm{S})$$

gen	total generation in the subtree $\mathcal{G}_{\text{sub}}$, of which the current considered block/ cut vertex is the root,
f	total flow into the port,
S	set of edges that are switched in $\mathcal{G}_{\text{sub}}$.

Note that a label stores a set of switched edges and when no edge is switched, we set $\mathrm{S} = \emptyset$. This is because later on, we also combine labels and then we can simply form the set union of their switched edges. With this approach we do not have to store two separate kinds of labels. Furthermore, let $\mathscr{L}$ be the finite set of all possible labels. Then we can define a function $L\colon V \to \mathcal{P}(\mathscr{L})$, that stores a set of labels for a given vertex of $\mathcal{R}$. This means, we store a set of labels at blocks and cut vertices.

The exponential algorithm is listed in Algorithm 5.1 and an illustration of its execution is given in Figure 5.2. It starts by creating the rooted BC-tree $\mathcal{R}$ of the network $\mathcal{N}$ and then invokes the ComputeMTSF procedure.

Algorithm 5.1: Exponential Algorithm

Input: Network $\mathcal{N} = (G, V_G, V_C, \mathsf{cap}, b)$
Output: Tuple $(\mathsf{gen}, \mathsf{S})$, where gen is the accumulated generation of $\mathcal{N}$ induced by the set of switched edges S, such that: $\mathsf{gen} = \mathrm{MPF}(\mathcal{N} - \mathsf{S}) = \mathrm{MTSF}(\mathcal{N})$.

1 $\mathcal{BC} := (G, V_G, V_C, \mathsf{cap}, b, \mathsf{B}, \mathsf{C}) \leftarrow$ BCTree($\mathcal{N}$)
2 $\mathcal{R} := (G, V_G, V_C, \mathsf{cap}, b, \mathsf{B}, \mathsf{C}, v_r, \mathsf{parent}, \mathsf{children}) \leftarrow$ RootedBCTree($\mathcal{BC}$)
3 **return** ComputeMTSF($\mathcal{R}$)

Procedure ComputeMTSF

Input: Rooted BC-tree $\mathcal{R} = (G, V_G, V_C, \mathsf{cap}, b, \mathsf{B}, \mathsf{C}, v_r, \mathsf{parent}, \mathsf{children})$
Output: Tuple $(\mathsf{gen}, \mathsf{S})$, where gen is the accumulated generation of $\mathcal{R}$ induced by the set of switched edges S, such that: $\mathsf{gen} = \mathrm{MPF}(\mathcal{R} - \mathsf{S}) = \mathrm{MTSF}(\mathcal{R})$.

1 **forall** $\beta \in (\mathsf{B} \cup \mathsf{C})$ *in reverse level-order* **do**
2 $L(\beta) \leftarrow \emptyset$ // initialize L
3 **if** $\beta \in \mathsf{B}$ **then**
4 $(\mathsf{gen}, \mathsf{S}) \leftarrow$ ComputeBlockLabels($\mathcal{R}, \beta$)
5 **else**
6 $\{\gamma_1, \ldots, \gamma_n\} \leftarrow \mathsf{children}(\beta)$
7 $\Pi \leftarrow \prod_{i=1}^{n} L(\gamma_i)$
8 **forall** $(l_1, \ldots, l_n) \in \Pi$ **do**
9 $(\mathsf{gen}_i, \mathsf{f}_i, \mathsf{S}_i) \leftarrow l_i$ // $l_i \in (l_1, \ldots, l_n)$
10 $\mathsf{f}' \leftarrow \sum_{i=1}^{n} \mathsf{f}_i$
11 $\mathsf{gen}' \leftarrow \sum_{i=1}^{n} \mathsf{gen}_i$
12 $\mathsf{S}' \leftarrow \cup_{i=1}^{n} \mathsf{S}_i$
13 $L(\beta) \leftarrow L(\beta) \cup \{(\mathsf{gen}', \mathsf{f}', \mathsf{S}')\}$
14 **return** $(\mathsf{gen}, \mathsf{S})$

This procedure iterates over the vertices of $\mathcal{R}$ in *reverse level-order*. In each iteration, we make the distinction between blocks and cut vertices. For blocks, we invoke the ComputeBlockLabels procedure, which is discussed in the next passage. It yields a tuple (gen, S) of generation gen and set of switched edges S, which is the result of computing an MTSF on $\mathcal{R}$.

For cut vertices, initially, the Cartesian product Π of labels from all n children is computed. Then, for each combination of labels $(l_1, \ldots, l_n) \in \Pi$, the flow and generation from each label l_i is added up into f' and gen', and we form the set union S' of switched edges S_i (Line 9 to end). Lastly, we create a new label with the latter three attributes and store it at that cut vertex. The final step of the procedure is to return the tuple of (gen, S) provided by the ComputeBlockLabels procedure.

Procedure ComputeBlockLabels

Input: Rooted BC-tree $\mathcal{R} = (G, V_G, V_C, \mathsf{cap}, b, \mathsf{B}, \mathsf{C}, v_r, \mathsf{parent}, \mathsf{children})$, Block $\beta = (V, E, V_{cv})$

Output: A tuple containing the greatest accumulated generation gen of $\mathcal{R}$ and the set of switched edges S inducing gen.

```
1  {γ_1, ..., γ_n} ← children(β)
2  if n > 0 then
3      Π ← ∏_{i=1}^{n} L(γ_i)
4  else
5      Π ← {(0, 0, ∅)}                          // leaves have no incoming flow and generation
6  if β ≠ v_r then
7      forall π ∈ Π do
8          L(β) ← L(β) ∪ GenerateLabel(R, β, π, ∅)          // no switched edge
9          if |V(β)| > 2 then
10             forall e ∈ E(β) do
11                 L(β) ← L(β) ∪ GenerateLabel(R, β − {e}, π, e)
12         else
13             (gen, f, S) ← π                   // π is only one label for a bridge
14             {e_bridge} ← E(β)                 // Set E(β) contains one edge in this case
15             f' ← min(f, cap(e_bridge))        // adjust flow and gen of label
16             L(β) ← L(β) ∪ (gen, f', S)
17 else
18     Solution ← ∅
19     forall π ∈ Π do
20         Solution ← Solution ∪ RootCheck(R, β, π, ∅)
21         if |V(β)| > 2 then
22             forall e ∈ E(β) do
23                 Solution ← Solution ∪ RootCheck(R, β − {e}, π, e)
24     if Solution ≠ ∅ then
25         return arg max_gen Solution
26 return (0, ∅)          // no feasible solution found → no generation and switching
```

Within this procedure, the Cartesian product of labels from the children is computed and stored in Π. Since this procedure is invoked only for blocks, children can only be cut vertices. If the block does not have children, this means there is no incoming power flow nor incoming generation. Then we set Π to just be a set of a single label with zero generation and flow, as well as no switched edges, yielding $\{(0, 0, \emptyset)\}$ (Line 4).
Next, for all non-root blocks, we consider every combination $\pi \in \Pi$. Note that here the term π is used as we are not interested in a specific label of that combination. This means the notation $(l_1, \ldots, l_n)$ and π both refer to a combination of labels.
For each π, we first generate a label for when no edge is switched (Line 7) and store it for that block. Then we check if the current block is a *bridge*, i.e. a block containing only one edge. If it is not, then we switch every edge once and generate a label, which we then store for that block. In the other case, we know that there is only one child. Therefore, a combination can only consist of a single label, which we can then retrieve (Line 12). We do not switch the edge, as it would break the circuit. But we do need to adjust the flow

and generation of the label, as it could be limited by the capacity of the bridge.
For the root we iterate over every combination and initially consider the case where no edge is switched. Here and in the following step, we do not store a label, but we invoke a separate RootCheck procedure, that yields the MPF for a given set of switched edges. The solution is stored in the set *Solution* (Line 19). Then, if the root is not a bridge, we switch every edge once and store the possible solutions. If the set *Solution* is not empty, we return the solution with the greatest generation and this is the MTSF of $\mathcal{N}$. After every combination is considered, we return the tuple $(0, \emptyset)$ which means, there is no generation and no switching.

Procedure GenerateLabel

Input: Rooted BC-tree $\mathcal{R} = (G, V_G, V_C, \mathsf{cap}, b, \mathsf{B}, \mathsf{C}, v_r, \mathsf{parent}, \mathsf{children})$,
Block $\beta = (V, E, V_{cv})$,
π: combination of labels,
$\mathsf{e}_{\text{switched}}$: switched edge

Output: Set of a single label, which holds the generation gen, outgoing flow f and switched edge of β. If no feasible flow exists, an empty set is returned.

```
1 if F ← MPF(N(β, π)) then
2     f, gen ← 0
3     gen ← Σ_{(gen_i, f_i, S_i) ∈ π} gen_i + Σ_{(g,w) ∈ E(β)} F(g, w)      w ∈ V(β), g ∈ V_G(N(β, π))
4     f += Σ_{(u,v) ∈ E(β)} F(u, v)          u ∈ V(β), v ∈ parent(β)
5     S ← ⋃_{i=1}^{n} S_i          (gen_i, f_i, S_i) ∈ π, |π| = n          // unify switched edges
6     S ← S ∪ {e_switched}
7     return {(gen, f, S)}
8 else
9     return ∅
```

In the GenerateLabel procedure, we first compute the MPF for the block. Since an MPF is computed on a network, we need to transform the block into one. Thereby, given a block β and a combination π of labels, we can represent β as a network by adding a generator for every label of that combination with positive generation and a load otherwise. The cut vertex with that label then becomes either a generator or load. The resulting network is denoted by $\mathcal{N}(\beta, \pi)$ (Line 1). If there exists no MPF, we return $\emptyset$. If it does exist, we collect the generation *gen* of a block by adding up the flow of every edge incident to a generator and add it to the overall generation in the subtree of the current block. Note that generators and consumers reside in their own block and thereby cannot interfere with their generation/demand.
Afterwards we collect the flow at the port f in the same way, but here we consider incident edges to the port. Then, for each label l_i in the combination π, we form the set union S of its switched edges $S(l_i)$ (Line 3). Finally, we also insert the edge, that was currently switched into that union. Then we can return a set with one tuple (gen, f, S). We need to return a set here and not just the tuple, because we return an empty set if no MPF for that block exists.

Procedure RootCheck

Input: Rooted BC-tree $\mathcal{R} = (G, V_G, V_C, \mathsf{cap}, b, \mathsf{B}, \mathsf{C}, v_r, \mathsf{parent}, \mathsf{children})$,
Block $\beta = (V, E, V_{cv})$,
π: combination of labels,
$\mathsf{e}_{\text{switched}}$: switched edge

Output: Set of a single tuple containing the accumulated generation $\mathsf{gen}_{\text{total}}$ of $\mathcal{R}$ and set S of switched edges inducing $\mathsf{gen}_{\text{total}}$. If no feasible flow exists, an empty set is returned.

```
1  gen_total ← 0
2  if F ← MPF(N(β, π)) then
3      S ← ⋃_{i=1}^{n} S_i            (gen_i, f_i, S_i) ∈ π, |π| = n
4      R' ← R − S
5      forall v_c ∈ V_cv(β) do
6          f_{v_c} ← Σ_{(u,v)∈E(β)} F(u, v_c)        u ∈ V(β)        // outgoing flow at v_c
7          L(v_c) ← (0, f_{v_c}, ∅)                                  // associate flow to cut vertex
8          π_sub ← L(v_c)
9          N_sub ← N(SUBTREE(R', v_c))                               // network of subtree rooted at v_c
10         if F_sub ← MPF(N_sub, π_sub) then
11             gen_total += Σ_{(g,w)∈E(N_sub)} F(g, w)        w ∈ V(N_sub), g ∈ V_G(N_sub)
12         else
13             return ∅
14     return {(gen_total, {S ∪ e_switched})}
15 else
16     return ∅
```

The final procedure is RootCheck of which the purpose is to compute the MPF with a given combination π firstly for the root and secondly for the rest of the network, given the set of switched edges. This means that for every block a switching possibility has been chosen and we now want to compute the total generation for the network.
Initially we compute the MPF for the root block where the given edge is switched and unify all switchings, as specified by the given combination π, into one set S (Line 3). The edges of S are then switched within our rooted BC-tree $\mathcal{R}$ and the result is assigned an alias $\mathcal{R}'$ (Line 4). Then at every cut vertex v_c of the root, we determine the resulting outgoing flow f_{v_c} (Line 6) and assign it to v_c in the form of a single combination (Line 7), which is then retrieved as an alias π_{sub} (Line 8). Now we can also obtain the subtree $\mathcal{N}_{\text{sub}}$ rooted at cut vertex v_c (Line 9), compute the MPF for it with π_{sub} and accumulate the generations of $\mathcal{N}_{\text{sub}}$ into the total generation gen_{total} (Lines 10 and 11). If an MPF does not exist, we can stop early by returning an empty set (Lines 13 and 16). The idea here is that f_{v_c} determines how much power a subtree has to provide overall. And by computing an MPF for it, we can verify if that amount can be supplied and if so, how much generation is needed. After all cut vertices have been considered, we have accumulated the generation for the whole network and we can return it along with the set of all switched edges in a tuple (Line 14).

The complexity of the algorithm is exponential in the total number of block of $\mathcal{R}$. This is due to considering every combination, where at most one edge is switched within a block. And the upper bound for the number of all possible combinations is given by $\prod_{\beta \in \mathrm{B}} |E(\beta)| + 1$.

Since a switching is represented by a label, a way to decrease the number of all considered combinations is to decrease the labels stored with each block/ cut vertex. In the next section, we are going to pick up on this topic again.

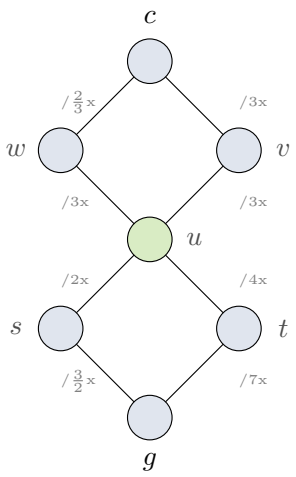
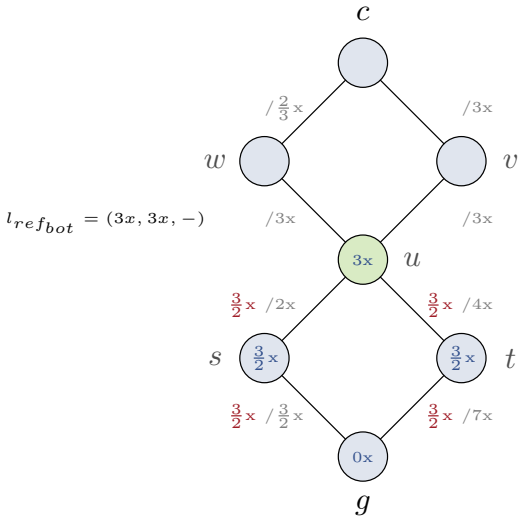
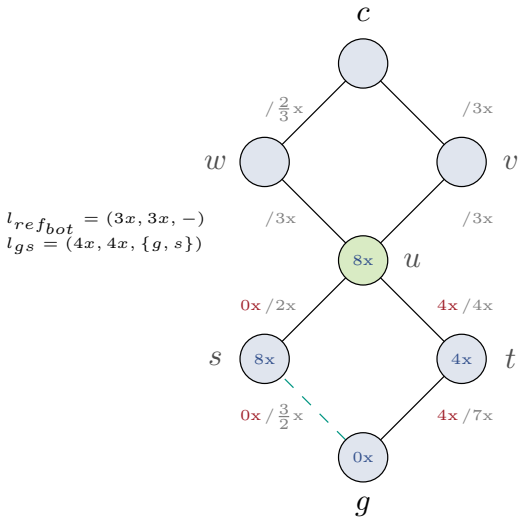


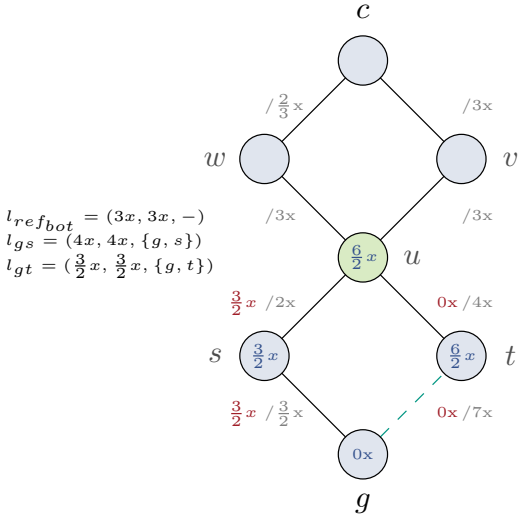
Figure 5.2: Illustration of the exponential algorithm on a cactus graph with one generator g and one load c. The line capacities and flows are written next to the edges in ■ and ■ respectively. Phase angles are written into the vertices in ■. The BC-tree of the graph has two blocks and one cut vertex ■, the upper block is set to be the root.



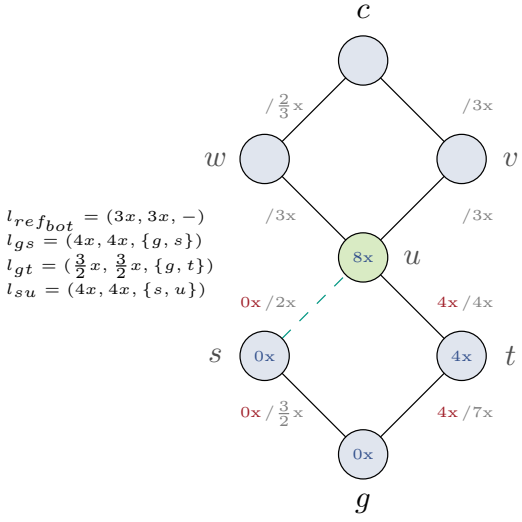
(a) Starting with the lower block when no switching occurred. The flow into u as well as the generation at generator g is $3x$ and the corresponding label is stored at u.



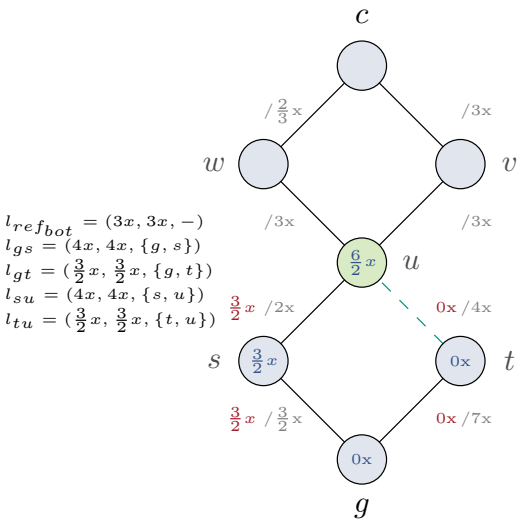
(b) The edge $\{g, s\}$ is switched ■. There is no flow on the edge $\{s, u\}$ and the path (g, t, u) exhibits a flow of $4x$.



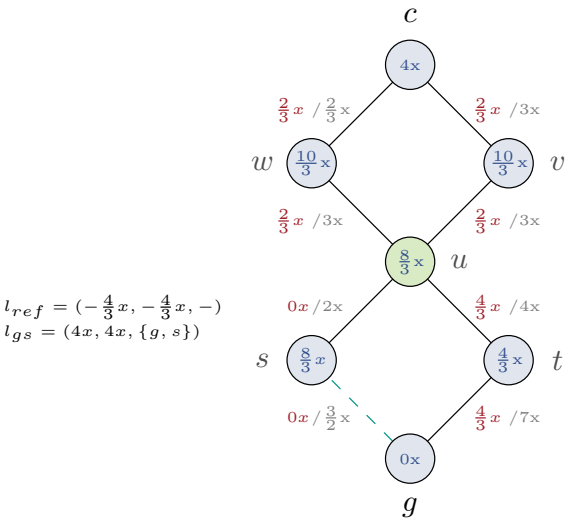
(c) The edge $\{g, t\}$ is switched and the resulting flow is $\frac{3}{2}x$.



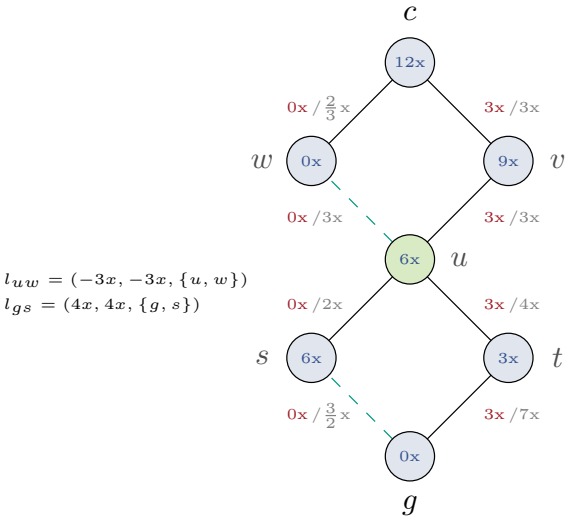
(d) Switching edge $\{s, u\}$ yields the same result as $\{g, s\}$.



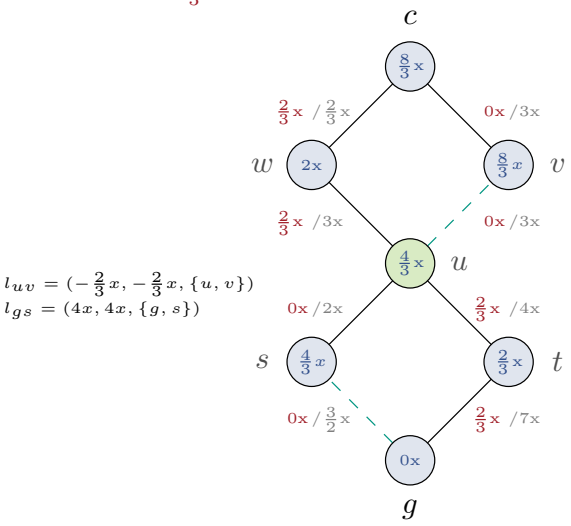
(e) Switching edge $\{t, u\}$ yields the same result as $\{g, t\}$.



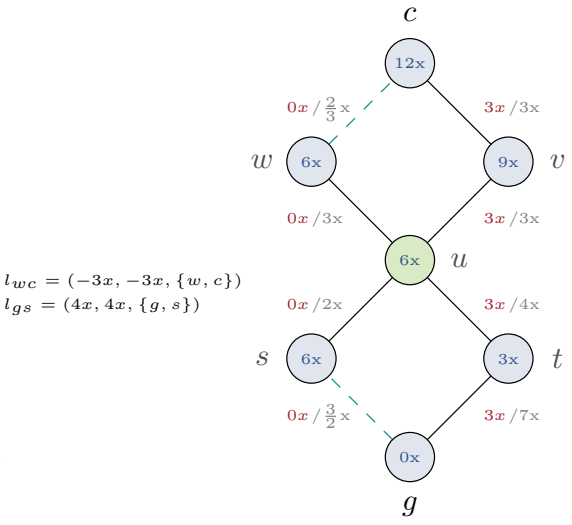
(f) Now the upper block is reached, that is the root. The label with the highest generation is chosen, and with it, the switching is carried out for the root. Without switching, we obtain an *MPF* of $\frac{4}{3}x$.



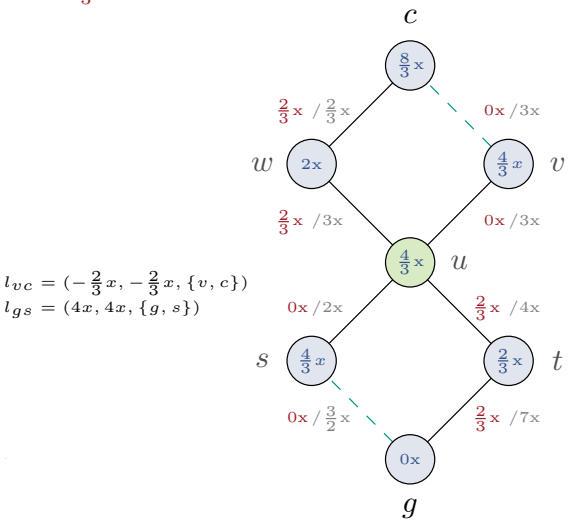
(g) The next switched edge is $\{u, w\}$. Here we get an increased *MPF* of $3x$.



(h) Switching edge $\{u, v\}$ decreases the *MPF* down to $\frac{2}{3}x$.



(i) The *MPF* is the same as for $\{u, w\}$, when switching edge $\{w, c\}$.



(j) Lastly, the edge $\{v, c\}$ is considered. The resulting *MPF* is not greater than the previous ones. Now, all switchings have been considered and the one exhibiting the highest *MPF* is chosen. Both, $\{\{g, s\}, \{w, c\}\}$ and $\{\{g, s\}, \{u, w\}\}$ yield the same *MPF* of $3x$, therefore either one can be used as a solution.

5.2 Runtime Improvements

Here we introduce an improvement that can be made to decrease the overall running time of the exponential algorithm. Having to iterate over every combination of labels is the limiting factor here, as there is an exponential magnitude of labels. Therefore it seems intuitive to work out techniques to reduce the overall amount of considered combinations.

5.2.1 Label Buckets

Now we are not taking into consideration the exact flow value of a label, instead we *round* this value according to a given parameter. By doing so, we can group *similar* labels in terms of flow. Then for every group, if two labels share the same generation, we can chose one of them arbitrarily and dismiss it.
We achieve this by introducing *buckets* to store labels in, which means we now store a set of buckets instead of a set of labels. A bucket is essentially a container with a key or identifier κ, that stores labels: for our purposes the key will be the flow of a label. We also provide a parameter $\epsilon > 0$ which determines the range or *interval* of labels that are stored in this bucket. More precisely, a label $l = (\text{gen}, \text{f}, \text{S})$ is assigned to bucket b_i with $i \in \{-n, \dots, n\}$ and key κ_i, such that

$$\kappa_i = \begin{cases} a, & f \geq 0 \\ b, & \text{otherwise.} \end{cases}$$

a and b are defined as

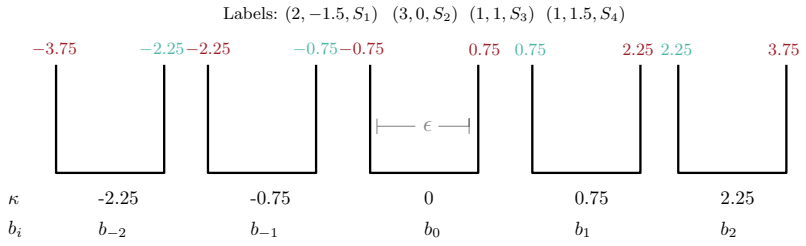
$$a = \begin{cases} 0, & 0 \leq f < \frac{\epsilon}{2} \\ (2i+1)\frac{\epsilon}{2}, & (2i+1)\frac{\epsilon}{2} \leq f < (2i+3)\frac{\epsilon}{2} \end{cases}$$

$$b = \begin{cases} 0, & -\frac{\epsilon}{2} < f < 0 \\ -(2i+1)\frac{\epsilon}{2}, & -(2i+3)\frac{\epsilon}{2} < f \leq -(2i+1)\frac{\epsilon}{2} \end{cases}$$

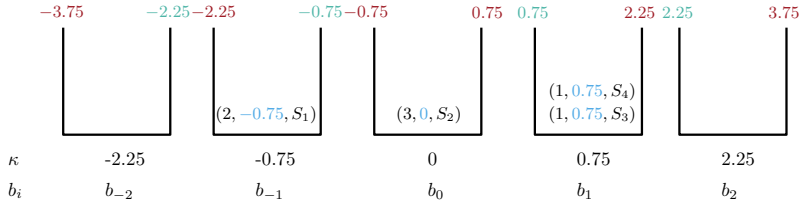
Intuitively we divide the range of possible flow values into sections of size ϵ whereby each label falls into the range of one section. Then the key of a bucket becomes the flow of the labels stored in it. The key is either 0, the left interval bound in case of positive flows or the right interval bound in case of negative flows. We use this policy because this way, we can ensure a feasible flow after we have modified the flow values of labels. It is also worthy to note here that we do not just start at key 0 and allocate buckets by incrementing/decrementing by ϵ. If we would do so, then there would be two buckets of range $(-\epsilon, 0]$ and $[0, \epsilon)$, both of which include 0. This means any two labels that are assigned to either bucket end up with a flow of 0. Also these buckets would then cover a range of 2ϵ in contradiction to the definition.
From this we can specify an upper bound for the index of buckets n for a given network $\mathcal{N}$:

$$n = \left\lceil \frac{\text{MF}(\mathcal{N})}{\epsilon} \right\rceil$$

The maximum flow MF is an upper bound for the MTSF of $\mathcal{N}$, hence there are no labels with a greater flow. The same holds for negative flows: in this case, there is no smaller flow. For a visualization of this endeavour see Figure 5.4.



(a) 5 buckets are allocated with $\epsilon = 1.5$ and $\mathrm{MF}(\mathcal{N}) = 7$ for four generated labels, that are not yet assigned. ■ marks an included and ■ marks an excluded interval bound.



(b) Now the labels are assigned to buckets and bucket b_2 is empty. ■ marks the adjusted flow. Note that one label in bucket b_1 can be dismissed as they are sharing the same generation of value 1.

Figure 5.4: Bucket illustration

Once a label has been placed into the appropriate bucket, the key of a bucket becomes its flow value while the generation remains the same. So we are limiting the information of labels, which is the consequence of rounding the labels.

Introducing buckets to the exponential algorithm requires a single adaption. We only have to replace the set of labels at each block/cut vertex with a set of buckets and when generating a label, we need to make sure that it gets placed into the appropriate bucket. Other steps such as creating combinations of labels remain the same.

5.3 Implementation

The implementation of the exponential algorithm and its improvement were realized in the programming language *C++* version 14. In order to compute a solution for the MILP of an MPF, the MILP-*solver* Gurobi optimizer version 8.0.1. [Gur16] has been chosen. No optimizations as far as parallelization have been made, the algorithm was implemented as it is.

6. Evaluation

In this chapter we evaluate the performance of the exponential algorithm (ExA). We start by evaluating the plain algorithm as it is and then we do the same, but with the runtime improvement added from the previous section.

In order to evaluate the performance we have considered six test cases representing power grids as cacti. These are modified IEEE test cases from which edges were manually deleted until the resulting graph was a cactus. There existed vertices, being both generator and consumer, in the original cases. As this would have contradicted our model, we proceeded to remove its load. Each case was then further transformed in a way that generators and consumers are placed at degree 1 vertices, as is described in Section 4.3. This was also done, even when generators and consumers already were degree 1 vertices. The specifications for each case after this transformation can be seen in Table 6.1 below.

IEEE Case Nr.	|V|	|E|	$\|V_C\|$	$\|V_G\|$
4	8	8	2	2
9	15	15	3	3
14	23	23	6	5
30	50	53	16	6
39	65	69	19	9
57	89	94	32	5

Table 6.1: Specifications for the modified IEEE cases as cacti after generators and consumers have been placed at degree 1 vertices. The total number of vertices, edges, consumers and generators are listed from left to right.

With these cases at hand we could then conduct performance tests. The hardware used consisted of an Intel i5, four core processor operating at a clock of 2.53 GHz and 8 GB of physical RAM. We then successively executed the algorithm with every test grid as input and different key figures were determined. The first remark to be made here is that the MILP results of both MPF and MTSF, as well as the ExA, were identical. This means that for every case, switching was not beneficial, which is a bit of an unsatisfying result. But these were the only available test cases and therefore we can already mark generating reasonable test cases for cacti as a first part of future work.
The first figure we are going to examine are running times in terms of duration, where

we start out with the overall running time and then look at individual times for each procedure.

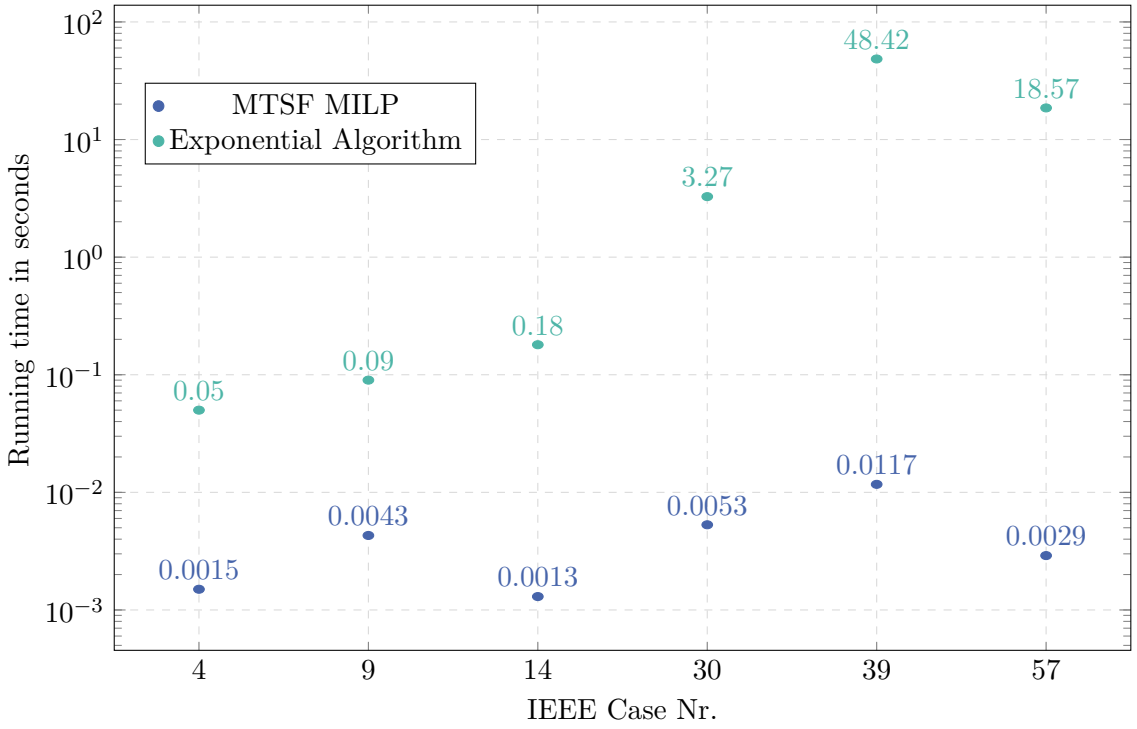


Figure 6.1: Running times plotted of the exponential algorithm and the MTSF MILP for every modified IEEE case.

In Figure 6.1 the total running times of the MTSF MILP, as well as the ExA are plotted for every test case and we can observe that the latter runs slower than the former. Both running times reach their maximum at case 39, whereas their difference is greatest for case 57 with an order of magnitude 10^3 seconds. We also notice a non linear increase for the ExA with every case up until case 39, where it decreases. The MILP does not quite follow the same pattern, e.g at case 14 it drops to its minimum and at case 57 it is less than at case 30, whereas the running time of the ExA is greater at case 57 than at case 30.

Now we are going to take a look at the overall amount of processed labels along with the specifications of each BC-tree, in order to draw a conclusion on the structure of the test cases and to make sense of the progressions of the plot.

IEEE Case Nr.	|B|	No. of Bridges	|C|	Total No. of Labels
4	5	4	4	10
9	10	9	9	38
14	19	17	12	85
30	44	40	26	561
39	51	46	36	15673
57	79	74	47	6167

Table 6.2: Specifications for the BC-trees of modified IEEE cases as cacti after generators and consumers have been placed at degree 1 vertices. The total number of blocks, bridges, cut vertices and labels are listed from left to right. Note that the number of bridges is also contained in the total number of blocks.

In Table 6.2 the specifications of the BC-trees, along with the total number of labels generated are listed. We can immediately observe a correlation between the total running time and the number of labels. The progression of the latter is remarkable, as it roughly increases fourfold at case 9 and then doubles at case 14. From there, it increases fivefold at case 30 and at case 39 by close to thirty times, where it reaches its maximum.
Since the number of combination is roughly given by the product of edges in a block plus one over every block, we can explain the drop in number of labels from case 39 to case 57, although the number of blocks, that are non bridges, stays the same. The number of labels then tells us, that the blocks in case 57 must be smaller in terms of number of edges, overall.

For the next part, we are going to examine the running times of each individual procedure, listed in Figure 6.2. Note that these running times do not sum up exactly to those of Figure 6.1 because instructions such as retrieving labels from BC-tree vertices are invoked by every procedure and were not considered here. However, the margin is negligible and the following remarks are still valid.

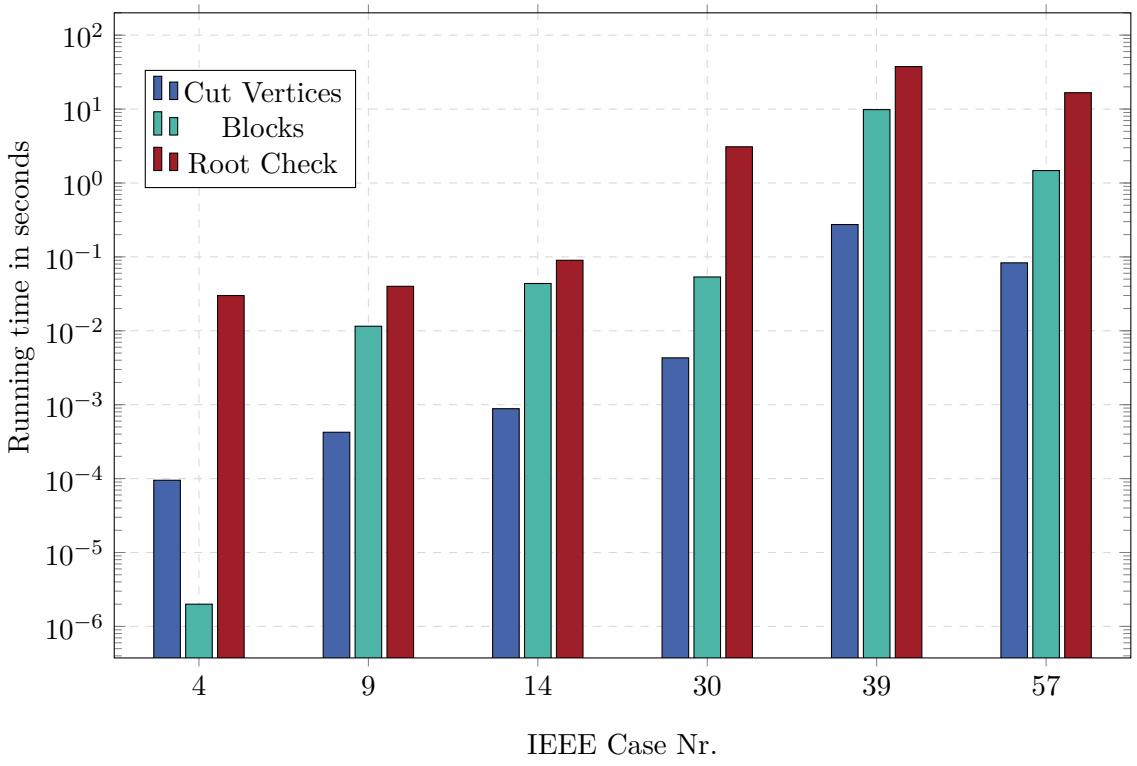


Figure 6.2: Individual running times plotted for every procedure of the exponential algorithm and for every modified IEEE case. The bars ■ and ■ refer to the statements executed in the case differentiation of procedure ComputeMTSF (Line 3, Line 5).

We notice right away that procedure RootCheck is the most time consuming one and is the main cause for the overall running time, which is due to the large amount of combinations/labels considered within the root. This suggests that a decrease in labels to store is promising.
Furthermore, the second most time is spent within blocks, which is not true for case 4. The reason for that, is that there is only one non-bridge block of four edges, whereas there are also four cut vertices. For this block, we only iterate over the edges, but for cut vertices, we compute a Cartesian product with more overhead. From Figure 6.2 we observe that

as the block count, as well as the blocks in terms of number of edges, grow, processing a cut vertex becomes more time consuming. This also implies having to consider more combinations within a block, which brings us to the previous conclusion for a need to decrease the number of labels.

Now we are going to evaluate the algorithm when using buckets of varying range ϵ. We begin with $\epsilon = 0.001$ and then subsequently increase it to 0.1, 1 and 5. Note that these steps are chosen arbitrarily and finding the appropriate range requires fine tuning.

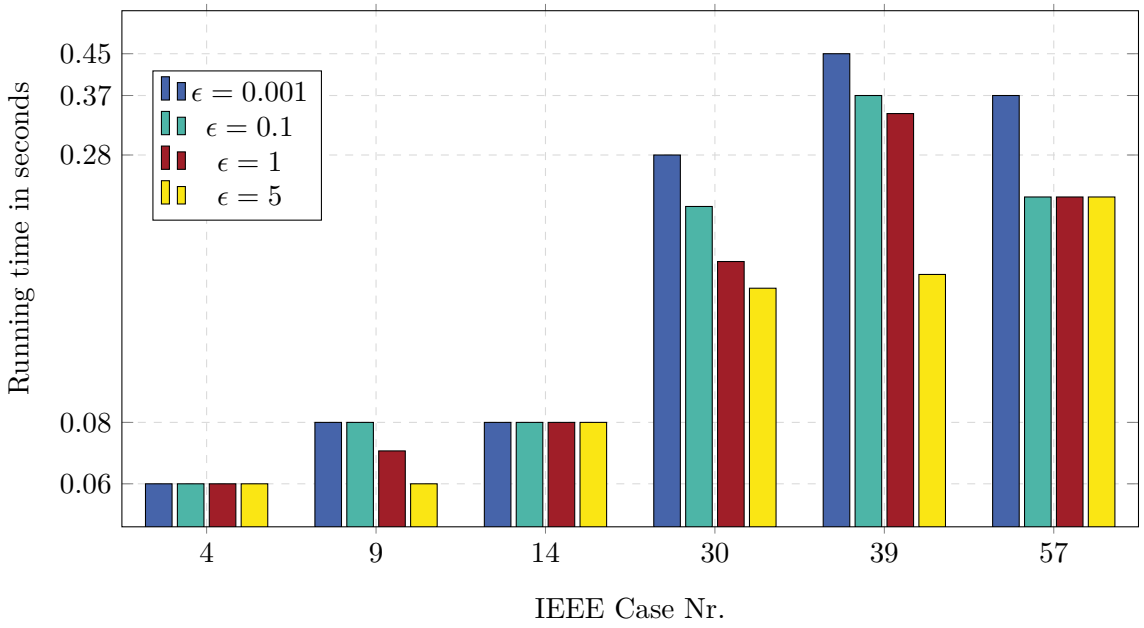


Figure 6.3: Total running times of the algorithm using buckets of variable size ϵ plotted for every modified IEEE case. Note that the scale for the running times is logarithmic.

		$\epsilon = 0.001$	$\epsilon = 0.1$	$\epsilon = 1$	$\epsilon = 5$
Case 4	Total No. of Labels	10	10	10	10
	Solution deviation in %	0	0	0	0
Case 9	Total No. of Labels	32	32	30	28
	Solution deviation in %	0	0	0	0
Case 14	Total No. of Labels	34	34	34	34
	Solution deviation in %	0	0	0	0
Case 30	Total No. of Labels	117	110	84	96
	Solution deviation in %	0	5.3	8.6	8.6
Case 39	Total No. of Labels	331	264	217	130
	Solution deviation in %	0	0.01	0.12	3.4
Case 57	Total No. of Labels	190	134	134	134
	Solution deviation in %	0	0	0	0

Figure 6.4: The total number of labels and solution deviation listed for ever modified IEEE case and different flow ranges ϵ.

One notices that the running times in Figure 6.3 have greatly decreased although the MTSF MILP still outperforms it. Whereas in Figure 6.1 for case 39 the algorithm finished in 48.42 seconds, it now takes 0.45 seconds at most. This is also the greatest speedup we have achieved amongst all test cases. Furthermore, we notice that increasing the bucket range decreases the running time. The progression of the plot overall follows Figure 6.1, except for $\epsilon = 5$ in case 57, where we can see an increase instead of a decrease. Also the low ranges of $\epsilon = 1$ and $\epsilon = 0.1$ yield the same running time and we can see in Figure 6.4 that the number of labels stays the same. An increase of ϵ creates new buckets, however labels are then not assigned into other buckets and none are dismissed. With this we notice that the most labels are dismissed, if we consider the range of flow Δf for the most labels with equal generation and then set $\epsilon = \Delta f$. However, this is just an estimation and the effectiveness of this would have to be proven.

Lastly, we consider Figure 6.4 where we see that the correlation between running times and the number of labels remains, except for case 30 where the number of labels increases for $\epsilon = 5$ to 96 compared to $\epsilon = 1$ with 84 labels.
The solution quality also progresses significantly well. Only for cases 30 and 39 we see a deviation, whereas in all other cases the optimal solution could be computed. With an increasing ϵ the solution deviation also increases, so there is a correlation between the two. This is due to the construction of labels, since the flow value f of a label gets rounded in a way, such that a flow with the given switching exists. It is also important to note here that for case 30 and $\epsilon = 1$ the solution deviation of 8.6 % seems large, however its optimal solution is at 1.5 and so the absolute solution deviation is not as large. This hints that there is a connection between ϵ and $\mathrm{OPT}_{\mathrm{MTSF}}(\mathcal{N})$, although research on that also has to be made.

7. Conclusion

In this thesis we formulated an algorithm that computes an optimal solution of MTSF for unbounded cacti networks. We achieve this by traversing its *Block-Cut-tree* in reverse level-order and solving instances of MPF in each block for every possible switching.

We introduced the idea of *labels* that are stored at every vertex of the BC-tree. They are used as input for the next traversal step, as they contain information on the MPF within a block such as flow into the parent vertex, as well as generation and switching in their subtrees. To represent partial solutions of greater subgraphs, labels are combined at cut vertices and the optimal solution for the whole graph is then determined at the root vertex.

We then observed that the algorithm exhibits an exponential complexity and identified generating all possible combinations of labels as the main cause. As a countermeasure we presented the *bucket* data structure to store labels with a flow range of ϵ. Within a bucket, for each occurring value of generation, only one label is stored. This leads to a greater or lesser reduction of overall labels, depending on the choice of ϵ, however, an optimal solution is then not guaranteed to be found.

Finally, we evaluated the algorithm for six power grids of IEEE cases, modified to be cacti, in comparison to solving the MILP of MTSF. We observed that the latter outperformed our algorithm at every case and noticed the greatest difference, when our algorithm took 48.42 seconds to compute an optimal solution, whereas the MILP took less than 0.012 seconds. Upon introducing buckets to the algorithm, we were able to greatly decrease its running time, however, we could not beat the MILP. For the same case and with a flow range of $\epsilon = 0.001$, it computed an optimal solution within 0.45 seconds and generated 331 labels, which is more than 15000 labels less. With $\epsilon = 1$, the amount was further reduced by more than 50 %, at a total of 84 labels. Here we also experienced the greatest deviation of 8,6 % from the optimal solution, however, the absolute deviation was not as large. For case 57, we were able to compute an optimal solution with the same flow range, hence the optimal ϵ differs from case to case.

7.1 Future Work

We presented an algorithm of exponential complexity as a first approach to solve MTSF on cacti using labels. A topic of future work is to look at other ways to store information of flow and generation within a block. This could be done by storing the flow in a block as a function of the generation in the subtree and then, every possible switching represents such a function. In order to dismiss other switchings, a criterion of *dominance* needs to be formulated here.
If the use of labels is continued, then one could investigate in ways of iterating the combinations at the root in a smarter fashion. As of now, every combination is considered here, however, if one solution is found, then we already know a lower bound for the generation and therefore, combinations with a generation that falls below this lower bound can be dismissed.

Regarding the use of buckets, research has to be made in determining the optimal flow range ϵ for a given network $\mathcal{N}$. For this the correlation between ϵ and $\mathrm{OPT}_{\mathrm{MTSF}}(\mathcal{N})$ has to be investigated in, as well as the range of flow of the most labels with equal generation, as was hinted in the evaluation part.

Another topic is to come up with a way of generating cacti from general test cases. The test cases used in this thesis were also not very interesting in terms of solution, as it turned out that switching was not beneficial for all of the cases. Therefore one would have to look into generating test cases, for which switching is interesting. Also with larger scale test cases, the bucket heuristic can be evaluated even further to gain more knowledge on how the solution quality is affected.

Lastly, we have only considered unbounded networks in this thesis, so designing an algorithm to solve MTSF for bounded cacti networks is also an open topic. To make this work, one would have to consider a redistribution of generation once the root block is reached. This is the case when only negative flows are passed into the root and as of now, this would lead to an overall generation of 0.
In this context the generation in a label also needs to be determined in a different way because as of now, it just represents an upper bound. It would also be interesting to see, if our algorithm can be extended, in order to solve MTSF on general graph classes and if so, what the necessary changes would be.

Bibliography

[Bau13] Heinz-Josef Bauckholt. *Grundlagen und Bauelemente der Elektrotechnik*. Lernbücher der Technik. Hanser, München, 7., aktualisierte aufl. edition, 2013.

[BB11] Clayton Barrows and Seth Blumsack. Optimal transmission switching analysis and marginal switching results. In *IEEE Power and Energy Society General Meeting*, pages 1–3, 2011.

[BBB13] Clayton Barrows, Seth Blumsack, and Russell Bent. Using network metrics to achieve computationally efficient optimal transmission switching. In *46th Hawaii International Conference on System Sciences (HICSS)*, pages 2187–2196, 2013.

[Bra68] D. Braess. Über ein paradoxon aus der verkehrsplanung. *Unternehmensforschung*, 12(1):258–268, 1968.

[FOF08] Emily B. Fisher, Richard P. O'Neill, and Michael C. Ferris. Optimal transmission switching. *IEEE Transactions on Power Systems*, 23(3):1346–1355, 2008.

[GRW+18] Alban Grastien, Ignaz Rutter, Dorothea Wagner, Franziska Wegner, and Matthias Wolf. The maximum transmission switching flow problem. In *Proceedings of the Ninth International Conference on Future Energy Systems*, e-Energy '18, pages 340–360, New York, NY, USA, 2018. ACM.

[GSO12] J. Duncan Glover, Mulukutla S. Sarma, and Thomas J. Overbye. *Power System Analysis and Design*. Cengage Learning, fifth edition edition, 2012.

[Gur16] Gurobi Optimization, Inc. *Gurobi optimizer reference manual*, 2016. `http://www.gurobi.com`, Accessed: 2017-01-14.

[KM80] Hans-Jürgen Koglin and Hermann Müller. Overload reduction through corrective switching actions. In *International Conference on Power System Monitoring and Control*, volume 24, pages 159–164, 1980.

[LGH14] Karsten Lehmann, Alban Grastien, and Pascal Van Hentenryck. The complexity of DC-switching problems. *CoRR*, abs/1411.4369, 2014.

[LGH15] Karsten Lehmann, Alban Grastien, and Pascal Van Hentenryck. The complexity of switching and FACTS maximum-potential-flow problems. *CoRR*, abs/1507.04820, 2015.

[NW99] George L. Nemhauser and Laurence A. Wolsey. *Integer and combinatorial optimization*. Wiley-interscience series in discrete mathematics and optimizationA Wiley-interscience publication. Wiley, New York, NY, 1999.

[OCS04] Thomas J. Overbye, Xu Cheng, and Yan Sun. A comparison of the ac and dc power flow models for lmp calculations. In *Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04)*

- *Track 2 - Volume 2*, HICSS '04, pages 20047.1–, Washington, DC, USA, 2004. IEEE Computer Society.

[PBL+12] M. G. Pala, S. Baltazar, Peng Liu, Hermann Sellier, B. Hackens, F. Martins, Vincent Bayot, X. Wallart, L. Desplanque, and Serge Huant. Transport Inefficiency in Branched-Out Mesoscopic Networks: An Analog of the Braess Paradox. *Physical Review Letters*, 108, February 2012.

[PP97] Eric I. Pas and Shari L. Principio. Braess' paradox: Some new insights. *Transportation Research Part B: Methodological*, 31(3):265–276, 1997.

[Rah17] Md. Saidur Rahman. *Basic Graph Theory*. Undergraduate Topics in Computer ScienceSpringerLink : Bücher. Springer, Cham, 2017.

[Sch18] Reinhard Scholz. *Grundlagen der Elektrotechnik : eine Einführung in die Gleich- und Wechselstromtechnik*. Fachbuchverlag Leipzig im Carl Hanser Verlag, München, [2018].

[Uni93] University of Washington. 14-bus-diagram, 1993. `https://www2.ee.washington.edu/research/pstca/pf14/pg_tca14fig.htm`, Accessed: 2018-09-15.

[Uni99] University of Washington, Department of Electrical Engineering. *Power systems test case archive*, 1999. `https://www2.ee.washington.edu/research/pstca/`, Accessed: 2017-11-14.

[Wes01] Douglas B. West. *Introduction to graph theory*. Prentice Hall, Upper Saddle River, NJ, 2. ed. edition, 2001.

[WT12] Dirk Witthaut and Marc Timme. Braess's paradox in oscillator networks, desynchronization and power outage. *New Journal of Physics*, 14(8):083036, 2012.

[ZMST11] Ray D. Zimmerman, Carlos E. Murillo-Sanchez, and Robert J. Thomas. Matpower: Steady-state operations, planning, and analysis tools for power systems research and education. *IEEE Transactions on Power Systems*, 26(1):12–19, 2011.