Karlsruhe Institute of Technology

Minimum-Cost Flow Algorithms for the Wind Farm Cabling Problem

Bachelor Thesis of

Marc Jenne

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Sascha Gritzbach
	Matthias Wolf

Time Period: 1st December 2019 – 30th April 2020

Selbstständigkeitserklärung

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, August 5, 2020

Abstract

The Wind Farm Cabling Problem (WCP) describes the problem of finding a suitable cabling between a set of turbines and substations in wind farms with the goal of transmitting the whole electricity produced by the turbines to the substations. While there exist numerous possibilities of feasible cablings, it is of interest to find the one that minimizes the total costs of the required cables.
This problem can be modeled as a Minimum-Cost Flow Problem (MCF), a well-known problem for which multiple algorithms already exist. These algorithms are proven to find an optimal solution in polynomial time; however, they cannot be used for the WCP without adaptions because of the primary difference between both problems: while the cost function is linear in the MCF, it is a non-linear step function in the WCP.

In this thesis, we take a closer look at some of those algorithms and examine whether they can be adapted so that they are able to provide solutions for the WCP. We describe the algorithms and their different approaches to solve the MCF and outline the problems that arise once the cost function becomes non-linear. We successfully develop an adaption of the Successive Shortest Path Algorithm and provide multiple strategies to solve the WCP with this algorithm. In the following experimental evaluation, we compare these strategies among each other and then compare the best variant to other existing WCP algorithms in terms of running times and quality of the found solutions.

Deutsche Zusammenfassung

Das Windfarm-Verkabelungsproblem (WCP) beschreibt das Problem, für Windfarmen eine geeignete Verkabelung zwischen einer Menge von Turbinen und Substationen zu finden. Das Ziel einer solchen Verkabelung ist der Transport der gesamten von den Turbinen produzierten Elektrizität zu den Substationen. Während zahlreiche solcher zulässigen Verkabelungen möglich sind, sind wir daran interessiert, unter diesen die kostengünstigste zu finden.
Dieses Problem kann als Minimum-Cost Flow Problem (MCF) modelliert werden, ein bekanntes Problem, für das bereits zahlreiche Algorithmen existieren. Für diese Algorithmen ist bewiesen, dass sie in polynomieller Zeit optimale Lösungen finden. Ohne weitere Anpassungen können diese jedoch nicht für das WCP verwendet werden, da sich beide Probleme vor allem in einem Aspekt voneinander unterscheiden: Während die Kostenfunktion im MCF linear ist, verwendet das WCP eine nichtlineare stufenförmige Kostenfunktion.

In dieser Arbeit betrachten wir einige dieser Algorithmen genauer und untersuchen, ob sie so angepasst werden können, dass sie Lösungen für das WCP finden können. Wir beschreiben die Algorithmen und ihre verschiedenen Ansätze, das MCF zu lösen, und analysieren die Probleme, die sich durch eine nichtlineare Kostenfunktion ergeben. Wir entwickeln erfolgreich eine Anpassung des Successive Shortest Path Algorithm und stellen verschiedene Strategien vor, das WCP mit diesem Algorithmus zu lösen. In der folgenden experimentellen Auswertung vergleichen wir zunächst diese Strategien untereinander, um dann die beste Variante mit anderen, bereits existierenden WCP-Algorithmen im Hinblick auf Laufzeit und Lösungsqualität zu vergleichen.

Contents

1. Introduction

1.1 Motivation

In times of climate change and discussions on renewable energies, wind energy holds an important place. In 2019, 15% of the electricity demand in the European Union was covered by wind power. It is currently the second largest power source and has been estimated to outpace natural gas, the current number one, in a few years [win].
A considerable part of wind energy is produced in offshore wind farms. They consist of turbines, generating electrical energy, and substations, where the generated energy is collected and subsequently transmitted to land. Generated energy is transported from the turbines to the substations by a set of cables, although it is also possible to connect turbines among each other. Usually, more than one cable type is available, differing in the capacity and in the costs: a cable that can carry more units of electricity has higher costs. The cabling between turbines and substations is called *internal cabling* and is only one part in the process of planning a wind farm; there is also the problem of positioning the turbines and substations, as well as the connections from the substations to the onshore stations. Once the last two decisions have been made, the internal cabling is left; and of course one is interested in finding the minimum-cost cabling. Given fixed positions, produced units of electricity per turbine and capacities of the substations, we call that problem of finding the minimum-cost internal cabling the Wind Farm Cabling Problem (WCP).
Since the WCP is $\mathcal{NP}$-hard [Sta18], finding optimal solutions takes a lot of time on large instances. This is arguably a problem in the dynamic process of planning a wind farm, for example when considering many different positionings. In this process of planning, it is impractical to wait hours to calculate the optimal cabling for just one possible positioning. For this reason there is definitely an interest in finding feasible internal cablings in a reasonable time, even if that means finding a trade-off between computation time and the quality of the provided solution.

The WCP can be modeled as a flow network, where the energy produced by the turbines represents the flow that has to be transported to the substations, respecting the capacity constraints of the cables and the substations. The problem of finding the minimum-cost flow is known as the Minimum-Cost Flow Problem (MCF) and has already been examined in detail, offering various algorithms to solve the problem. Nevertheless, there is one difference between the WCP and the MCF, namely the cost function. In the MCF, each edge has a linear cost function, meaning each additional unit of flow along this edge causes a constant cost, regardless of the previous amount of flow on that edge. In contrast,

the costs in the WCP depend on the used cable type. While more units of flow may need a cable type with a higher capacity, is does not matter if the capacity of a used cable is saturated; once it is in use, the costs for it have already been paid and it is irrelevant whether there is capacity left or not. Therefore, sending more flow along an edge causes more costs only if a larger cable type has to be used, otherwise the additional flow on that edge is free of costs. We refer to such a non-linear cost function as a *step cost function*.
Despite of that difference, treating the WCP as a modified MCF is valid. Thus a possible approach to tackle the WCP is to modify existing algorithms for the MCF in a way to solve (i.e. compute feasible solutions) the WCP.

1.2 Related Work

1.2.1 The Wind Farm Cabling Problem

One of the first approaches to deal with the WCP uses a decomposition of the problem into three layers [BVMO16]. Two of these layers map to well-known graph problems, whereas the third layer is solved by a greedy algorithm. After this work, several approaches of solving the WCP have been developed, both optimal and non-optimal variants. For optimal solutions, Mixed-Integer Linear Program (MILP) variations can be used [LR13]. The downside of these is the running time: while computing solutions on small instances is possible in a reasonable time, it can take many days on instances with up to 500 turbines. In contrast, the WCP can be solved by metaheuristics, for example Simulated Annealing [LRWW17]. While not being optimal, they provide a good trade-off between running time and solution quality.
A quite different approach is to model the WCP as a flow problem and use known techniques for solving it as a minimum-cost flow problem. One already examined way uses negative cycle canceling [GUW$^+$18] and is proven to compute non-optimal, but still good solutions in short running times: while the MILP solver Gurobi took more than one day, the negative cycle canceling approach terminated in under two minutes on several instances with up to 500 turbines.

1.2.2 The Minimum-Cost Flow Problem

In contrast to the quite new WCP, the MCF is much older and has been studied extensively. The first pseudo-polynomial time algorithm for this problem was the Out-of-Kilter Algorithm which was independently developed by Minty [Min60] and Fulkerson [Ful61] in 1960 and 1961, respectively. Another early approach was the Successive Shortest Path Algorithm, developed independently by Busaker and Gowen [BG60], Iri [Iri60] and Jewell [Jew62]. In 1972, Edmonds and Karp [EK72] were the first to solve the MCF in weakly polynomial time. They found that using vertex potentials ensures nonnegative edge lengths, resulting in faster shortest path computations. The first algorithm that solved the MCF in strongly polynomial time was developed in 1985 by Tardos [Tar85]. In 1997, a polynomial time primal network simplex algorithm for the MCF that runs in $O(V^2 E \log(VC))$ (with C as the maximum edge cost) was published by Orlin [Orl97].
Beside those mentioned algorithms, many others have been developed, most of them working with similar approaches or based on existing ones. All of them, as well as the algorithms mentioned above, provide optimal solutions for the MCF.

Since the asymptotic worst-case running time does not represent the actual performance of an algorithm, experimental studies that compare MCF algorithms in practice are of interest. One of the first computational studies was performed in 1974 by Glover et al. [GKK74]. Important contributions to the practical evaluations were achieved 1993 in the First DIMACS Implementation Challenge [JM93]. More recent implementations and computational analyses were done by Kovács and Király [KK12][Kov15] in 2015.

1.3 Contributions

In this thesis, we look at four existing MCF algorithms: the Successive Shortest Path Algorithm, the Primal-Dual Algorithm, the Out-Of-Kilter Algorithm and the Relaxation Algorithm. For each of them, we point out which approaches it uses to solve the minimum-cost flow problem. Thereafter we examine if they can be adapted in a way to solve the WCP. For the Successive Shortest Path Algorithm, the only successful adaptable algorithm, we provide various strategies and evaluate them in terms of solution quality and running time. The best strategy is then compared to two existing WCP algorithms, namely an exact MILP solver and a negative cycle canceling approach. The other three algorithms are analyzed with respect to the problems arising due to the step cost function of the WCP, and we outline why they are not appropriate to be further adapted.
For each algorithm, we also examine a special case of the WCP where only one cable type is available.

1.4 Outline of the Thesis

The first chapter gave an overview about the problem, the current state and what we are approaching in this thesis. In Chapter 2, we build a theoretical foundation of the concepts and notations that are used in the following chapters.
With this knowledge, we examine in the third chapter four existing algorithms for the minimum-cost flow problem. For each of those algorithms there are two parts. The first part presents the original algorithm and its basic concepts. In the second part, we emphasize the problems that occur as a result of the WCP step cost function and provide ideas how to deal with them.
In Chapter 4 we evaluate our adapted Successive Shortest Path Algorithm in terms of running times and quality of the provided solutions. First, we compare the different strategies of our algorithm, and afterwards, the best variant is compared to two other existing WCP algorithms.
Chapter 5 then concludes the results and provides an outlook on further ideas.

2. Preliminaries

2.1 Flow Networks

As seen in the previous chapter, the minimum-cost flow problem algorithms examined in the following all work on flow networks. Although we assume the reader to be familiar with that topic, we briefly recap the definition of flows and flow networks and set our notations and assumptions that are used in the following. The concepts are based on [AMO93, Ch. 9], the formal notations sometimes differ to match the later introduced wind farm model.

A flow network is a directed graph $G = (V, E)$ with a *cost* function $c : E \to \mathbb{N}$ and a *capacity* function $u : E \to \mathbb{N}$. We shortly denote $c((i, j))$ as c_{ij} for an edge $(i, j) \in E$, and in an analogous way we refer to $u((i, j))$ as u_{ij}. The total costs of an edge (i, j) are linear in the amount of flow on this edge, with c_{ij} denoting the costs for one unit of flow on (i, j). With each vertex $i \in V$, a number $b(i) \in \mathbb{Z}$ is associated to represent the supply (if $b(i) > 0$) or demand (if $b(i) < 0$) of the vertex. A vertex with $b(i) = 0$ is called a transshipment vertex.
A flow is a function $x : E \to \mathbb{N}$, and indicates how much flow is transmitted along an edge. Just like above, we refer to the flow on an edge $(i, j) \in E$ as x_{ij}. A flow is called feasible if it satisfies two constraints. The first one is called *mass balance constraints* (Equation 2.1) and states that for each vertex, the total outgoing flow minus the ingoing flow has to equal its supply or demand, respectively. The second one is called *flow bound constraints* (Equation 2.2) (also referred to as nonnegativity and capacity constraints) and states that the amount of flow on each edge has to be nonnegative and may not exceed the edge's capacity.

$$\sum_{j \in V:(i,j) \in E} x_{ij} - \sum_{j \in V:(j,i) \in E} x_{ji} = b(i), \quad \forall i \in V. \tag{2.1}$$

$$0 \leq x_{ij} \leq u_{ij}, \quad \forall (i, j) \in E. \tag{2.2}$$

We are now able to describe the MINIMUM-COST FLOW PROBLEM (Equation 2.3): among all feasible flows, find the one that minimizes the total flow costs, referred to as $z(x)$:

$$\text{Minimize } z(x) = \sum_{(i,j) \in E} c_{ij} \cdot x_{ij}, \text{ subject to the constraints 2.1 and 2.2.} \tag{2.3}$$

Some of the algorithms we examine later use the concept of *pseudoflows*. A pseudoflow x is a flow that satisfies the nonnegativity and capacity constraints (Equation 2.2), but may

violate the mass balance constraints (Equation 2.1). For that concept, we introduce the so called *imbalance* for each vertex of the graph, that is, how much ingoing or outgoing flow is required to satisfy the mass balance constraint for that vertex. Formally, we define the imbalance of a vertex i as

$$e(i) = b(i) + \sum_{j \in V:(j,i) \in E} x_{ji} - \sum_{j \in V:(i,j) \in E} x_{ij}, \quad \forall i \in V. \tag{2.4}$$

We call i an excess vertex if $e(i) > 0$, and we call i a demand vertex if $e(i) < 0$. A vertex i with $e(i) = 0$ is called *balanced*.

Now we take a look at the following assumptions from [AMO93, As. 9.1-9.5], for which we state that they all do not restrict the generality since they can be fulfilled by transformations.

Assumption 1. *All data (cost, supply/demand, and capacity) are integral.*

Assumption 2. *The network is directed.*

Assumption 3. *The supplies/demands at the vertices satisfy the condition* $\sum_{i \in V} b(i) = 0$ *and the minimum cost flow problem has a feasible solution.*

Assumption 4. *We assume that the Network G contains an uncapacitated direct path [...] between every pair of vertices.*

Assumption 5. *All [edge] costs are nonnegative.*

Finally, as the further presented algorithms rely on that, we recap the concept of *residual networks*. For a given flow x, the residual network $G(x)$ is defined in the following way: each edge $(i, j) \in E$ is replaced by two edges, (i, j) and (j, i). The cost of (i, j) remains c_{ij}, since sending one unit of flow along (i, j) means to increase the flow on it. In contrast, sending one unit of flow along (j, i) amounts to canceling one unit of the existing flow on (i, j). This decreases the flow cost by c_{ij}, and therefore the cost of edge (j, i) is defined as $c_{ji} = -c_{ij}$. Furthermore, both edges have a *residual capacity*, referred to as r_{ij} or r_{ji}, respectively. The residual capacity is the remaining capacity of an edge, i.e. the remaining amount of flow that can be transmitted without exceeding the edge's capacity. Thus, $r_{ij} = u_{ij} - x_{ij}$ and $r_{ji} = x_{ij}$. The residual network contains only edges with positive residual capacities, i.e. in case an edge becomes saturated, it drops out of $G(x)$.

2.2 Wind Farm Model

In this chapter, we describe the modeling of a wind farm that is used in the following, as well as the formal description of the Wind Farm Cabling Problem as it is described in [GUW+18]. Furthermore, we point out some differences between the wind farm model and the previously described "classic" flow networks.
The vertices of a wind farm are divided into two subsets: V_T, representing the turbines, and V_S, representing the substations. Of course, $V_T \cap V_S = \emptyset$, and we define $V = V_T \cup V_S$ as the overall vertices of the wind farm. In a real wind farm, there is no direction of a cable between two turbines or between a turbine and a substation. For modeling reasons, as we have to obtain a directed graph, the direction of each connection is chosen arbitrary. With these definitions, we have the Graph $G = (V, E)$ modeling the wind farm vertices and connections.
As flow can still be transmitted along both directions of an edge, the flow on an edge (i, j) is interpreted in the following way: if $x_{ij} > 0$, x_{ij} units of flow are sent from i to j. If

$x_{ij} < 0$, $-x_{ij}$ units of flow are sent from j to i.
We assume each turbine to produce exactly one unit of electricity; going back to the flow network model, that is $b(i) = 1, \forall i \in V_T$. While we can see the turbines as the supply vertices, the wind farm does not contain real demand vertices. Instead, each substation has a capacity, modeling how many units of flow it can store. That capacity is assigned by the function $cap_{sub} : V_S \to \mathbb{N}$. Thus, in contrast to the demand in the origin flow network, the substations just have an upper bound, but no need for any amount of flow.
Now we head on to the greatest difference between the both models, the cost function. We recap that in the flow network model each edge is assigned a cost, representing the costs of sending one unit of flow along that edge. In contrast, the edges (i.e. the possible connections between vertices) of the wind farm all have the same cost function as they can all use the same set of cables. Each cable type is determined by two properties: its capacity $cap_{cab} \in \mathbb{N}$ and its cost per unit length $c_{cab} \in \mathbb{R}_{\geq 0}$. The set K is defined as all available cable types represented by a pair (cap_{cab}, c_{cab}) plus the cable types $(0, 0)$ and (∞, ∞). The latter ones represent that no cable is used on an edge or that no cable has a large enough capacity, respectively. With the cable types, we are now able to define the cost function (that counts, as mentioned before, for *each* edge) as $c : \mathbb{Z} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ with

$$c(x) = min\{c_{cab} : (cap_{cab}, c_{cab}) \in K, |x| \leq cap_{cab}\} \quad \forall x \in \mathbb{Z}. \tag{2.5}$$

That means, the cheapest cable type with sufficient capacity to carry $|x|$ units of flow is chosen.
Since the costs of a cable are not only determined by the cable type, but also by the length (recap that c_{cab} are the costs *per unit length*), we also have a function $len : E \to \mathbb{R}_{>0}$, representing the geographic distance between two turbines (or a turbine and a substation). With that, the costs of sending $|x|$ units of flow along an edge $e = (i, j)$ can be denoted as $cost(x, e) = c(x) * len(e)$.
With those definitions, the wind farm is modeled as a network $\mathcal{N} = (G, V_T, V_S, len, cap_{sub}, c)$. Similar to the original flow network, a flow x on a wind farm is feasible if it satisfies the following constraints. The first ones match to the mass balance constraints (Equation 2.1) and state that no flow may remain in a turbine (Equation 2.6) and that the net flow in substations may not exceed their capacity (Equation 2.7). The last constraints ensure that there is no outgoing flow at any substation (Equation 2.8).

$$\sum_{j \in V:(i,j) \in E} x_{ij} - \sum_{j \in V:(j,i) \in E} x_{ji} = 1, \quad \forall i \in V_T. \tag{2.6}$$

$$\sum_{j \in V:(j,i) \in E} x_{ji} - \sum_{j \in V:(i,j) \in E} x_{ij} \leq cap_{sub}(i), \quad \forall i \in V_S. \tag{2.7}$$

$$\begin{aligned} x_{uv} &\geq 0, \\ x_{vw} \leq 0, \quad \forall v \in V_S, \quad \forall (u, v), (v, w) &\in E. \end{aligned} \tag{2.8}$$

The total flow costs of a feasible flow in a wind farm are computed as $costs(\mathcal{N}, x) = \sum_{e=(i,j) \in E} c(x_{ij}) \cdot len(e)$.
The Wind Farm Cabling Problem can now be formulated as follows: among all feasible flows x on $\mathcal{N}$, find the one that minimizes $costs(\mathcal{N}, x)$.

2.3 Vertex Potentials and Reduced Costs

All of the later examined algorithms deal with the idea of node potentials (referred to as vertex potentials) and reduced costs. That concept is used for two reasons: to maintain nonnegative edge weights, allowing to solve shortest path problems quicker, and to prove the correctness of the algorithms. We summarize the idea and the most important points of the concept as found in [AMO93, pp. 307-310].

To understand where the idea of reduced costs has its origin, we first take a look at the shortest path problem. Having a start vertex s, a set of distance labels $d(\cdot)$ defines shortest path distances from s to each other vertex if two conditions are satisfied: first, the distances have to represent feasible distances (i.e. a path with this distance has to exist), and second the so-called shortest path optimality conditions:

$$d(j) \leq d(i) + c_{ij}, \quad \forall (i,j) \in E. \tag{2.9}$$

These conditions are very helpful for the validation of optimality: given a set of distances d or a set of paths, it is simple to check whether they define shortest paths, hence being optimal.
Equation 2.9 can be rewritten in an equivalent form:

$$c_{ij}^d = c_{ij} + d(i) - d(j) \geq 0, \quad \forall (i,j) \in E. \tag{2.10}$$

For an edge $e = (i,j)$ we can now interpret c_{ij}^d as an optimal reduced cost for e: it measures the cost of e relative to the shortest path distances $d(i)$ and $d(j)$. Given optimal distance labels, each edge has nonnegative reduced costs. Furthermore, $c_{ij}^d = 0$ if and only if the edge (i,j) is on a shortest path from i to j, and any shortest path from i to j uses only zero reduced cost edges.
To define *reduced costs* for more general minimum-cost flow problems, we first introduce *vertex potentials*: each vertex $i \in V$ is assigned a real number $\pi(i)$, and we call $\pi(i)$ the potential of i. With these potentials, we define the reduced costs of edge (i,j) as $c_{ij}^\pi = c_{ij} - \pi(i) + \pi(j)$ (see Equation 2.10). As the later algorithms use the reduced costs, it is important to keep the following property in mind:

Property 1. *For any directed path P from vertex k to vertex l, $\sum_{(i,j)\in P} c_{ij}^\pi = \sum_{(i,j)\in P} c_{ij} - \pi(k) + \pi(l)$.*

From Property 1 it results that using vertex potentials and reduced costs does not change the shortest path between two vertices k and l, since the length of every path between these vertices is increased by the same amount, namely $\pi(l) - \pi(k)$. Though, the computed lengths of paths may differ when using potentials.
From Property 1 also results the following property:

Property 2. *For any directed cycle W, $\sum_{(i,j)\in W} c_{ij}^\pi = \sum_{(i,j)\in W} c_{ij}$.*

With Property 2 and the *negative cycle optimality conditions*, which state that a feasible solution x^* is an optimal solution of the MCF if and only if the residual network $G(x^*)$ contains no negative cost cycle, we can now establish the *reduced cost optimality conditions*:

Theorem 2.1. *A feasible solution x^* is an optimal solution of the minimum-cost flow problem if and only if some set of vertex potentials π satisfy the following reduced cost optimality conditions:*

$$c_{ij}^\pi \geq 0 \text{ for every edge } (i,j) \text{ in } G(x^*). \tag{2.11}$$

Theorem 2.1 gives us the possibility to validate the optimality of solutions on the residual network. With the following theorem, we restate its meaning for the original network. That theorem is called the *complementary slackness optimality conditions.*

Theorem 2.2. *A feasible solution x^* is an optimal solution of the minimum cost flow problem if and only if for some set of vertex potentials π, the reduced costs and flow values satisfy the following complementary slackness optimality conditions for every edge $(i, j) \in E$:*

$$\text{If } c_{ij}^{\pi} > 0, \text{ then } x_{ij}^* = 0. \tag{2.12}$$

$$\text{If } 0 < x_{ij}^* < u_{ij}, \text{ then } c_{ij}^{\pi} = 0. \tag{2.13}$$

$$\text{If } c_{ij}^{\pi} < 0, \text{ then } x_{ij}^* = u_{ij}. \tag{2.14}$$

2.4 Further Notations

When discussing an algorithm's complexity and running time, we refer to the number of vertices of a graph as n and to the number of edges as m.

In some shortest path algorithms, we denote the edge weights as ω since these are general algorithms that do not necessarily work with reduced costs.

A path containing the edges $(a, b), (b, c), (c, d)$ is denoted as *a-b-c-d*.

In the graphical representation of wind farms, turbines are displayed as circles and substations as squares. For lack of space, the potential $\pi(i)$ and the imbalance $e(i)$ of a vertex i are shortened to π_i and e_i, respectively.

3. Examining the Algorithms

We now examine various minimum-cost flow problem algorithms. After describing them and underlining which approaches they use to solve the problem, we try to adapt them in a way that they are able to solve the WCP.

3.1 Successive Shortest Path Algorithm

3.1.1 Describing the SSP

We take a closer look on how the Successive Shortest Path Algorithm (SSP) works as it is described in [AMO93, pp. 320ff.].
The fundamental idea of the SSP is to iteratively solve shortest path problems until a feasible flow is achieved. In each step, the algorithm maintains a pseudoflow (section 2.1). By iteratively decreasing the imbalance (Equation 2.4) of the vertices, that pseudoflow is transformed into a feasible flow.
Before we take a look at the SSP in more detail, we introduce two vertex sets: we denote V_E as the set of all excess vertices and V_D as the set of all demand vertices. Note that the total excess always equals the total demand:

$$\sum_{i \in V_E} e(i) = -\sum_{i \in V_D} e(i). \tag{3.1}$$

The SSP, displayed in Algorithm 3.1, solves in each iteration (lines 5-12) a shortest path problem from an excess vertex to a demand vertex. This problem is computed on the residual network, with the reduced costs of each edge as edge weights. While finding a shortest path, the algorithm also computes and stores the shortest distances from the start vertex to each other vertex (line 7). These distances are then used to update the vertex potentials as seen in line 9.
When a shortest path, denoted as P (line 8), has been found, the maximum possible amount of flow, denoted as δ, is augmented along that path (lines 10-11). After that, the sets V_E and V_D and the reduced costs of each edge are updated (line 12), and the next step begins.

Let us take a look now at the correctness and the termination behaviour of the algorithm. We show that the algorithm always maintains a feasible pseudoflow at runtime and terminates with a feasible flow. At the beginning, the flow x is set to zero, which is a feasible pseudoflow since it does not violate any capacity constraints (Equation 2.2). In

Algorithm 3.1: Successive Shortest Path Algorithm

Input: Directed Network $G = (V, E)$, capacities u_{ij}, costs c_{ij}
Output: Optimal flow x^*

```
// Initialization
```
$x \leftarrow 0$
$\pi \leftarrow 0$
$e(i) \leftarrow b(i) \quad \forall i \in V$
$V_E \leftarrow \{i : e(i) > 0\}, V_D \leftarrow \{i : e(i) < 0\}$

```
// Main loop
```
while V_E *is not empty* **do**
  select a vertex $k \in V_E$ and a vertex $l \in V_D$
  compute shortest path distances $d(\cdot)$ from k to all other vertices in $G(x)$ with c_{ij}^{π} as edge weights
  $P \leftarrow$ shortest path from k to l
  $\forall i \in V : \pi(i) \leftarrow \pi(i) - d(i)$
  $\delta \leftarrow min[e(k), -e(l), min(r_{ij} : (i, j) \in P)]$
  augment δ flow along P
  update x, $G(x)$, V_E, V_D and all reduced costs

each step (until V_E and V_D are empty), the algorithm successfully finds an excess vertex and a demand vertex, because there can never be excess vertices without at least one demand vertex and vice versa, due to Equation 3.1. Because of using the reduced costs, all edge weights are nonnegative. Therefore, the shortest path distances to all vertices (with Assumption 4, a path to every vertex exists) are well defined and can be computed. With δ being always greater than zero, in each step at least one unit of flow is augmented from an excess vertex to a demand vertex. Thus, the total excess is decreased in each step. The algorithm terminates when no excess vertex is left, and because of Equation 3.1, there is also no demand vertex left. Hence, each vertex is balanced and the mass balance constraints (Equation 2.1) are satisfied, so a feasible flow is achieved.
Since the total excess is decreased in each iteration, the algorithm terminates after at most $n \cdot U$ iterations, where U denotes the highest excess among all vertices.

The running time of each iteration is dominated by the shortest path computation. With denoting the time taken by the latter as SP, the running time of the SSP is $O(n \cdot U \cdot SP)$. As mentioned before, the vertex potentials and the resulting reduced costs ensure that the shortest path computation has to handle only nonnegative edge weights. Thus, we can for example use Dijkstra's Algorithm [Dij59], running in $O(n \log n + m)$. With that, the overall running time of the SSP is $O(n \cdot U \cdot (n \log n + m))$.

3.1.2 Adapting the SSP

Before pointing out which parts of the original SSP need to be adapted for the WCP, we briefly recap some properties of our WCP model that help us for the development of solutions.
We have previously pointed out that the SSP augments as many units of flow as possible in each step from one excess vertex to one demand vertex. In the case of our wind farm model, every turbine is an excess vertex, and as every turbine produces exactly one unit of electricity, $e(i) = 1$ for each turbine i. One important difference between the minimum-cost flow modeling and the wind farm modeling is the absence of vertices with a fixed demand in our wind farm. The substations are the "destination" of the flow units, so all the flow from the turbines has to end up in substations, but they do not require a fixed amount

of flow; quite contrary, their only restriction is the capacity. As long as the capacity of a substation is not saturated, the substation *can* take more flow, but it does not have to. The problem with this property is that the SSP requires demand vertices, and these must have a fixed demand. To deal with it in our SSP for the WCP, we use the concept of a *supersubstation*. The supersubstation, referred to as vertex t, is an added vertex where all the demand is collected, i.e. in each step of the SSP, flow is augmented from one turbine to the supersubstation. It is created in the following way: for each substation i, we add an edge (i, t) from the substation to the supersubstation with $c_{it} = 0$ and $u_{it} = cap_{sub}(i)$. The demand of the supersubstation is set to $|V_T|$, and all substations are set to be balanced vertices. With that done, we have now modeled the substation capacities and solved the problem with the demand vertices without changing the costs of the final flow.

Now we head on to the greatest problem that occurs in our wind farm model in contrast to the standard minimum-cost flow problem: the non-linear cost function. As we have already seen, the reduced costs of an edge (i, j) in the SSP are computed as $c^{\pi}_{ij} = c_{ij} - \pi(i) + \pi(j)$. While c_{ij} is constant for each edge over the whole standard algorithm, it is not in the WCP. For the following idea on how to handle that problem, we briefly recap what is known about the augmented flow in each step: as the excess of each vertex is at most 1, we augment at most one unit of flow along an edge per step. With that in mind, in each step we can consider the costs c_{ij} for an edge as the costs of sending one additional unit of flow along that edge (Equation 3.2). That means, if the currently used cable on edge (i, j) has unused capacity, $c_{ij} = 0$ because we can send one more unit of flow without needing a more expensive cable type. If the currently used cable is saturated, the costs of sending one more unit of flow are computed as the difference in costs of the next in size cable type minus the current one.

$$c_{ij} = \text{costOfFlow}(\text{flowOnEdge}(i, j) + 1) - \text{costOfFlow}(\text{flowOnEdge}(i, j)). \tag{3.2}$$

In an analogous way, if the edge (i, j) has a negative flow (which means, that at the current moment we send flow along its reverse edge (j, i)), the costs c_{ij} are considered and computed as the saved costs if we send one unit of flow less along the edge (j, i).
With that idea, we can compute the reduced costs like in the original SSP, and could now run that modified algorithm (displayed in Algorithm 3.2) to solve the WCP. When the algorithm reaches the point of computing the shortest path distances, a new problem arises. As we remember, the vertex potentials and the resulting reduced costs do not change shortest paths (Property 1), but are used to ensure the nonnegativity of all edge weights, enabling us to compute the shortest distances very efficiently. With the modified costs (Equation 3.2), we lose that property, leading to negative edge weights and even negative cycles, as we can see in Figure 3.1. The problem here is that as soon as the distances are computed in the first iteration, c_{ij} is 1, because there is no flow on that edge, hence sending one unit requires a new cable type. Thus, the distance of j is 1. After sending one unit of flow along that edge, the new c_{ij} is zero, because sending one more unit of flow does not require a new cable type. But since the vertex potential of j is now -1, the reduced costs are negative. In the original SSP, that problem would not occur, because the value c_{ij} would never change between two iterations and therefore the reduced costs will always be nonnegative.

That leads to the question whether (and if so, how) we can now find the shortest path distances in a graph with possible negative edge weights and also possible negative cycles. An algorithm that can handle negative edge weights is the well-known Bellman-Ford Algorithm [Bel58]. However, in its common way, it is not going to solve our problem, because it detects negative cycles but it cannot compute shortest path distances in their presence. The reason is, that with a negative cycle there exists no shortest path as we can infinitely traverse that cycle, always decreasing the distance. What we need to find is the so-called

Algorithm 3.2: ADAPTED SUCCESSIVE SHORTEST PATH ALGORITHM FOR WCP

Input: Directed network $\mathcal{N} = (G, V_T, V_S, len, cap_{sub}, c)$
Output: Feasible flow x

```
   // Initialization
1  Add supersubstation t to N
2  x ← 0
3  π ← 0
4  V_E ← V_T
   // Main loop
5  while V_E is not empty do
6      select a vertex k ∈ V_E
7      compute shortest path distances d(·) from k to all other vertices in G(x) with
         c^π_ij as edge weights
8      P ← shortest path from k to t
9      ∀i ∈ V : π(i) ← π(i) − d(i)
10     augment one unit of flow along P
11     update x, G(x), V_E and all reduced costs
```

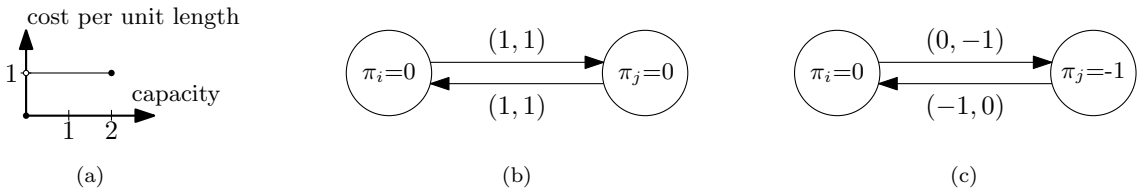

Figure 3.1: Negative reduced costs. The tuple on each edge denotes its costs and its reduced costs. Inside each vertex, its potential is displayed. (a) Step cost function; (b) Initial network; (c) Network after the first iteration and augmenting one unit of flow along (i, j).

Shortest Simple Path: the shortest path that does not visit a vertex more than once. This problem is known to be $\mathcal{NP}$-complete [Sch03, p. 114].
Therefore, we have to develop ways to find path distances that might not be the shortest, but "short enough" and that can be found in reasonable running time. Hence we are now willing to give up optimality, and head instead for using heuristics.

3.1.3 Finding Short Paths

3.1.3.1 Dijkstra

The first approach to find path distances is to use Dijkstra's Algorithm, although we know that it will not find the shortest path distances. Our implementation is displayed in Algorithm 3.3. To avoid negative cycles, an edge is not relaxed if its end vertex has already been visited (lines 10-11). To be able to check this, a list stores all vertices that have been removed from the priority queue (line 8). At this point, the optimality is not given anymore since we skip relaxations of some edges even if they would provide shorter distances.
In our implementation, Dijkstra runs in $O(n \log n + m)$ time and provides feasible distance and parent labels. In this context, *feasible* means that the paths induced by the parent labels exist and the distance labels represent distances along some paths from vertex k.

Algorithm 3.3: DIJKSTRA'S ALGORITHM

Input: Graph $G = (V, E, \omega)$, source vertex k
Data: Priority queue Q, List L of visited vertices
Output: Distances $\mathsf{d}(v)$ for all $v \in V$, path tree of k given by $\mathsf{parent}(\cdot)$

```
   // Initialization
 1 forall v ∈ V do
 2     d(v) ← ∞
 3     parent(v) ← null
 4 Q.INSERT(k, 0)
 5 d(k) ← 0
   // Main loop
 6 while Q is not empty do
 7     u ← Q.DELETEMIN()
 8     L.INSERT(u)
 9     forall (u, v) ∈ E do
10         if L.CONTAINS(v) then
11             continue
12         if d(u) + ω(u, v) < d(v) then
13             d(v) ← d(u) + ω(u, v)
14             parent(v) ← u
15             if Q.CONTAINS(v) then
16                 Q.DECREASEKEY(v, d(v))
17             else
18                 Q.INSERT(v, d(v))
```

From the parent labels we can extract a path from the start vertex k (i.e. the turbine we chose in that iteration) to the supersubstation.

3.1.3.2 A Modified Bellman-Ford

Another approach is the usage of a modified Bellman-Ford Algorithm (Algorithm 3.4) that does not create cycles. First, let us briefly refresh how Bellman-Ford works in the common implementation: it iterates $(n-1)$-times over all edges (lines 5-6), and if for an edge (u, v) holds that $d(u) + \omega(u, v) < d(v)$, that edge is relaxed (lines 9-11). Of course, this does not prevent negative cycles: if for example (u, v) and (v, u) both have negative costs, the distances of both u and v are decreased in each iteration and u and v form a cycle.
To ensure that those cycles are not created, our modified Bellman-Ford performs an additional check (lines 7-8) before an edge (u, v) is relaxed: if v is directly or indirectly (which means, with other vertices between) the parent of u, the edge is not relaxed, because that means a cycle would be created at this point. Algorithm 3.5 shows how that cycle detection works in detail: the parents of u, starting with its direct parent (line 1), are traversed backwards. If at some point we find v as the next parent, the algorithm returns that a cycle is detected. Otherwise, the algorithm terminates in case the current considered vertex has no parent (line 2), meaning that no cycle is detected (recall that all parents have been set to *null* at the beginning of the Bellman-Ford Algorithm 3.4).

That modified algorithm provides feasible path distances and parent labels, from which we can extract the required path from the start vertex to the supersubstation. However, it still does not provide optimal results, as we can see in Figure 3.2. For shortening

Algorithm 3.4: Bellman-Ford Algorithm

Input: Graph $G = (V, E, \omega)$, source vertex k
Output: Distances $\mathsf{d}(v)$ for all $v \in V$, path tree of k given by $\mathsf{parent}(\cdot)$

```
// Initialization
1 forall v ∈ V do
2     d(v) ← ∞
3     parent(v) ← null
4 d(k) ← 0
// Main loop
5 for i ← 0 to n − 1 do
6     forall (u, v) ∈ E do
7         if createsCycle(G, u, v) then
8             continue
9         if d(u) + ω(u, v) < d(v) then
10            d(v) ← d(u) + ω(u, v)
11            parent(v) ← u
```

Algorithm 3.5: Cycle detection

Input: Directed Network $G = (V, E)$, startvertex u, endvertex v
Output: True, if relaxation of edge (u, v) would create a cycle; false otherwise

```
1 currentParent ← u
2 while parent(currentParent) != null do
3     currentParent ← parent(currentParent)
4     if currentParent == v then
5         return True
6 return False
```

reasons, we ignore relaxations of edges where the start vertex has an infinite distance, since those relaxations do not have influence on the results. At the beginning of the algorithm, the distance of k is set to zero. In the first iteration, the edges (k, c) and (k, a) are relaxed. In the next iteration, the edges (a, b), (c, d) and, most important, (c, a) are relaxed because $-9 = d(c) + \omega(c, a) < d(a) = 1$. So now, vertex c is the parent of vertex a. Now in the next step, again (a, b) is relaxed, and the edge (b, c) would be relaxed, because $-98 = d(b) + \omega(b, c) < d(c) = 1$. At this point, the cycle detection checks if the end vertex c is somehow the parent of the start vertex b and realizes it to be true because of the path c-a-b. Thus, the relaxation is not performed since a cycle would be created. No more edges are relaxed in the algorithm after that. Now the computed shortest distance for vertex c is 1, with the path k-c. In fact, the real shortest simple path would have been k-a-b-c with a length of -98. Also, as quite obvious, the computed distance for d is not correct.

The running time of the original Bellman-Ford Algorithm is $O(n \cdot m)$. The added cycle detection check takes an extra time of $O(n)$ as it has to iterate over all vertices until no more parent is found in the worst case. With that, the modified algorithm runs in $O(n^2 \cdot m)$ time, which is much slower than Dijkstra's Algorithm, but still polynomial.

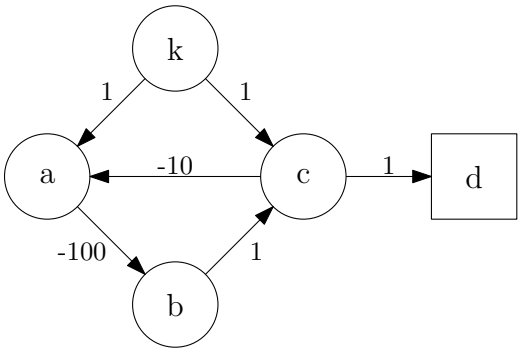

Figure 3.2: Example for non-optimal results with the modified Bellman-Ford Algorithm. On each edge, its weight is shown.

3.1.4 Running Time and Optimality

We now take a look at the running time of our developed algorithm and its solution quality from a theoretical view.

3.1.4.1 Running Time

In the previous subsections, two ways of finding paths were discussed, including their respective running times. The overall running time of our adapted SSP depends mostly on these path computations, as they are performed in each iteration. In the main loop, the algorithm iterates over each turbine, therefore it takes $O(n)$ iterations. The executed steps per iteration apart from finding the paths are solved in $O(n + m)$ time, which is less or equal to the runtime of all our possible shortest path algorithms, thus the latter is dominating the runtime during each iteration. With denoting the runtime of finding the shortest paths as $O(SP)$, our developed algorithm has an overall running time of $O(n \cdot SP)$.

3.1.4.2 Quality of the Solution

As we have seen, the quality of the solution (i.e. the total costs of flow) depends on the quality of the paths and distances we find with the chosen algorithm. However, in this subsection, we hypothesize that we are actually able to find the optimal shortest paths and examine the solution quality the algorithm provides under that assumption.
Considering each iteration separately, it is obvious that the algorithm finds the cost-optimal flow, as the flow costs are represented by the reduced costs and we made the assumption that we find the shortest path from the start vertex to the supersubstation. Therefore, in each step the algorithm finds the local optimum.
However, now we show that always finding the local optimum in each step does not always result in the global optimum and hence does not provide an optimal solution for the WCP. In Figure 3.3, a wind farm instance is displayed, with vertex d being the only substation. The supersubstation has been omitted in the representation. The length of each edge (i, j) can be seen in (b), as it equals its reduced costs c^{π}_{ij} at the beginning. In the first iteration, vertex a is chosen, and the shortest path distances are computed. As the shortest path from a to substation d is a-d, one unit of flow is augmented along there. In the second (third) step, vertex b (c) is selected as start vertex. The shortest path to d is the direct connection b-d (c-d), for this reason one unit of flow is augmented along it. The other possible path, namely b-a-d (c-a-d), has higher costs. After the third step, the algorithm terminates since there is no excess vertex left. In (e), the computed flow is shown: its costs are 10+15+15=40. That solution is not optimal, as we see in (f): the (feasible) flow here has costs 30+1+1=32.
The algorithm found the local optimum in each step, but still missed the global optimum.

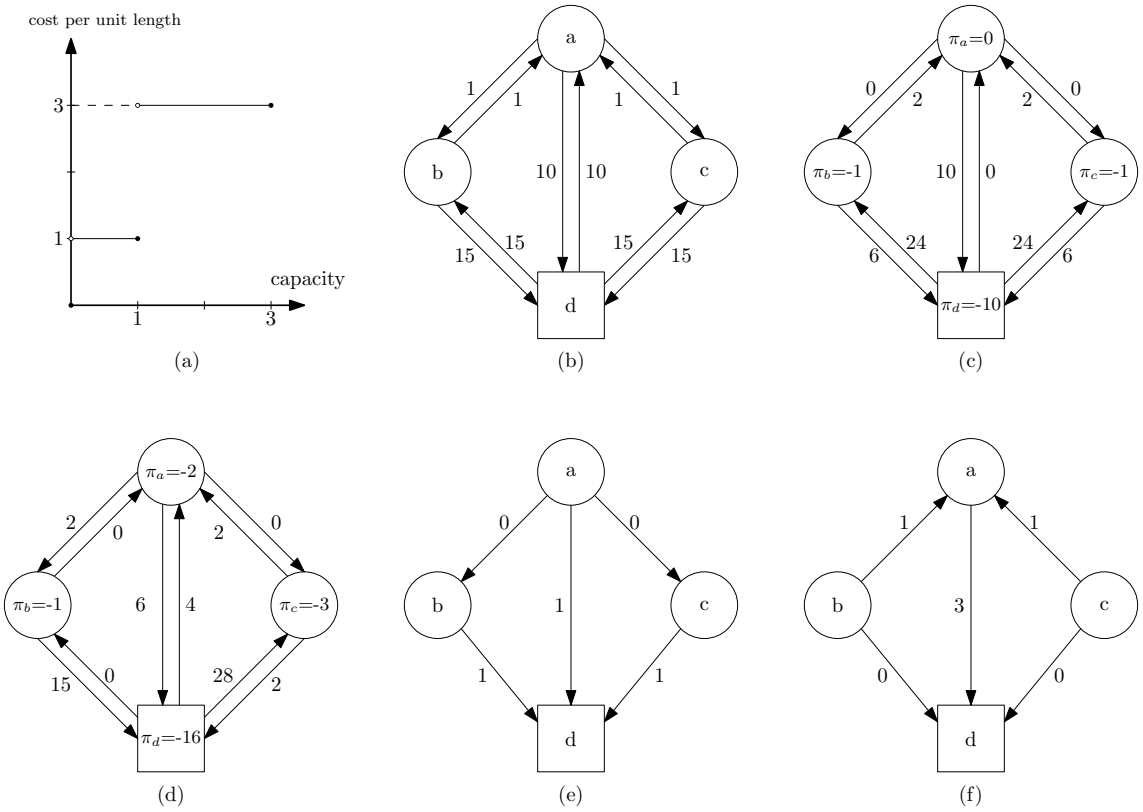

Figure 3.3: Non-optimal solution with the SSP. In (b)-(d), each edge is labeled with its reduced costs, inside each vertex its potential is displayed. In (e) and (f) the flow on each edge is shown. (a) Step cost function; (b) Initial network; (c) Residual network after the first step with a as start vertex; (d) Residual network after the second step with b as start vertex; (e) Final network flow after the third step with c as start vertex; (f) Network flow representing an optimal solution.

Due to the step function, the costs of edge (a, d) increased highly after augmenting the first unit of flow along it. That caused the edge not to be part of the shortest paths in the following steps, although using it two more times would lead to a better solution, since its high costs arise only in the second use while the third use would be "free".

3.1.5 SSP without Vertex Potentials

As mentioned before, the standard SSP uses vertex potentials to ensure that no negative costs occur and therefore Dijkstra's Algorithm can be used to compute the shortest paths. We have also seen that with step cost functions, negative costs do occur although we use vertex potentials. Since the upside of using potentials is not given anymore, another possible adaption of the SSP is to leave out the vertex potentials and reduced costs. Instead, as edge weights we use c_{ij} as computed in Equation 3.2. This further adaption requires only one change in Algorithm 3.2, namely taking out line 9 (updating all vertex potentials). Our analyses concerning the running time and quality of the solution still hold in that new variant, since we only altered the edge weights that are used when computing shortest paths. As in the first variant, we have two possibilities of finding paths: Dijkstra's Algorithm and our modified Bellman-Ford Algorithm. Similar to the first variant, both of these algorithms will not provide optimal solutions, since negative costs still occur in our new variant.

Totally, we have four variants of our modified SSP now: either using vertex potentials and reduced costs, or using the "normal" costs. For both strategies, we have two algorithms of finding short paths, Dijkstra and Bellman-Ford. We refer to those variants as DIJNOR (Dijkstra using normal costs), DIJPOT (Dijkstra using potentials and reduced costs), BELNOR (Bellman-Ford using normal costs) and BELPOT (Bellman-Ford using potentials and reduced costs).

3.1.6 Special Case - One Cable Type Only

When analyzing the adapted SSP, we figured out two points: in the first place, neither Dijkstra nor Bellman-Ford find the shortest paths due to the presence of negative edge weights. Having only one cable type does not solve this problem, since edge weights can still be negative. Secondly, the SSP was not optimal, even under the assumption that shortest paths can be computed correctly. In Figure 3.4 we observe that also with only one cable type the optimal solution is not always found. The length of each edge is displayed in the initial network in (b) and equals the initial costs. In the first step, c is selected as start vertex and one unit of flow is augmented along the shortest path, which is c-d. In the second step with b as start vertex, the shortest path is b-c-d, and the last iteration augments one unit of flow along a-b-d. The overall costs of the final flow (shown in (e)) are 36, while the optimal flow, displayed in (f), costs only 26.
Hence, the SSP does not benefit from having only one cable type.

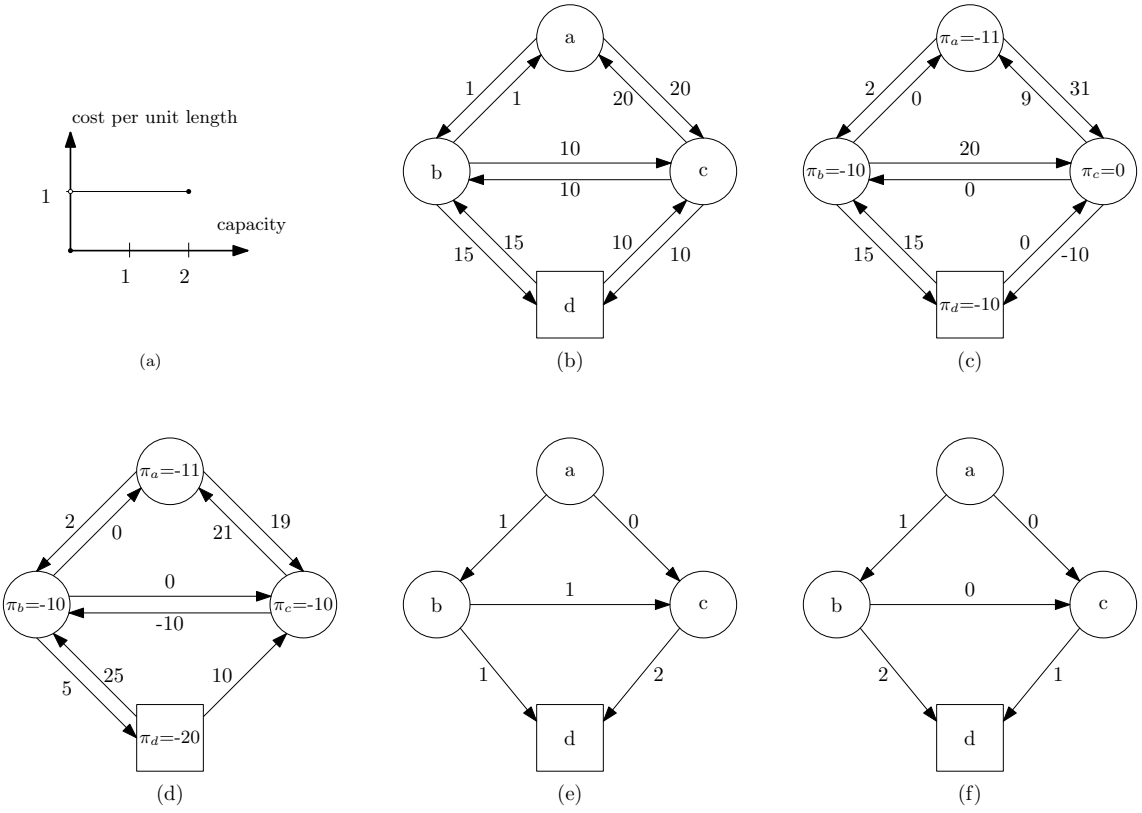

Figure 3.4: Non-optimal solution with only one cable type. In (b)-(d), each edge is labeled with its reduced costs, inside each vertex its potential is displayed. In (e) and (f) the flow on each edge is shown. (a) Step cost function; (b) Initial network; (c) Residual network after the first step with c as start vertex; (d) Residual network after the second step with b as start vertex; (e) Final network flow after the third step with a as start vertex; (f) Network flow representing an optimal solution.

3.2 Primal-Dual Algorithm

3.2.1 Describing the Primal-Dual Algorithm

The fundamental idea of the Primal-Dual Algorithm (PDA) [AMO93, pp. 324ff.] is similar to the SSP: it repeatedly solves shortest path problems and augments flow along these shortest paths until a feasible flow is achieved. In contrast to the SSP, in each iteration flow is not augmented along only one shortest path, but along *all* shortest paths by solving a maximum flow problem.
Before the main loop begins, the network is transformed in a way that there exists only one excess vertex and one demand vertex, respectively. This is achieved by a concept quite similar to the supersubstation we used earlier. There, we added a vertex that was connected with all demand vertices (namely the substations, in the case of WCP) so that all demand was collected in one vertex. This is exactly what the transformation does, with the new vertex referred to as *sink vertex t*. Analogous, a *source vertex s*, connected to all excess vertices, is added to the network where all excess is collected. More formally, the transformation performs the following steps:

1. Add source vertex s and sink vertex t to the network.
2. $\forall i \in V$ with $b(i) > 0$: add edge (s, i) with $u_{si} = b(i)$, $c_{si} = 0$.
3. $\forall i \in V$ with $b(i) < 0$: add edge (i, t) with $u_{it} = -b(i)$, $c_{it} = 0$.
4. $b(s) = \sum_{i \in V : b(i) > 0} b(i)$.
5. $b(t) = -b(s)$.
6. $\forall i \in V \setminus \{s, t\}$: $b(i) = 0$.

After that transformation, the PDA (Algorithm 3.6) works just like the SSP: while the source vertex s has excess left, the algorithm determines the shortest path distances from s to all other vertices, using the reduced costs as edge weights (line 6). With the distances, the vertex potentials are updated (line 7). The difference to the SSP is now the following: while the SSP augments as much flow as possible along one shortest path from the excess vertex to the demand vertex, the PDA augments flow along all shortest paths. To do so, the *admissible network* $G'(x)$ is defined as follows: $G'(x)$ consists of the same vertices as the residual network $G(x)$, but contains only those edges from $G(x)$ with zero reduced costs (line 8), since every shortest path includes only zero reduced edges. Now, a maximum flow problem from s to t is solved in $G'(x)$ and the computed flow is augmented (line 9). When that step of augmenting flow is done, the PDA again behaves like the SSP: the reduced costs as well as $e(s)$ and $e(t)$ are updated (line 10), and the next iteration begins.

The advantage of the PDA over the SSP is quite obvious: as it possibly augments flow along more than one path in each iteration, it is faster than the SSP (or, in the case of always finding just one shortest path, at least not slower).

3.2.2 Adapting the Primal-Dual Algorithm

As seen in the previous section, the Primal-Dual Algorithm differs from the SSP only in a single detail, namely the number of chosen paths. However, that detail has a huge impact on our adapted SSP. One underlying assumption of the latter has been that only one unit of flow is augmented in each step. With that, it was possible to determine the edge costs as the costs of sending one additional flow along that edge. That assumption is no longer given when using the PDA, because now more than one unit of flow can be augmented in each step. From that comes up the possibility of sending more than unit along one edge, and here arises a problem: how should the costs be determined when it is unknown how

Algorithm 3.6: Primal-Dual Algorithm

Input: Directed Network $G = (V, E)$, capacities u_{ij}, costs c_{ij}
Output: Optimal flow x^*

```
// Initialization
```
Transform network by adding s and t
$x \leftarrow 0$
$\pi \leftarrow 0$
$e(s) \leftarrow b(s)$, $e(t) \leftarrow b(t)$

```
// Main loop
```
while *e(s) > 0* **do**
  compute shortest path distances d from s to all other vertices in $G(x)$ with c_{ij}^{π} as edge weights
  $\forall i \in V : \pi(i) = \pi(i) - d(i)$
  define the admissible network $G'(x)$
  solve maximum flow problem and augment flow from s to t in $G'(x)$
  update x, $G(x)$, $e(s)$, $e(t)$ and all reduced costs

much flow is augmented there in a single step?
One possibility is to handle it as we did in the SSP, i.e. as the costs for sending exactly one additional unit of flow. It can be shown that in this case the algorithm would take wrong decisions, hence not finding the local optimum within one step. For illustration, we take a look at Figure 3.5. We assume each edge to have unit length. After transforming the flow network, we obtain the network shown in (b). On this network, the shortest path distances are computed and the vertex potentials (and therefore, the reduced costs) are updated. The resulting residual network is shown in (c), the bold edges belong to the admissible network that is now defined. On that, a maximum flow problem is solved. Since the outgoing edges from s all have a capacity of 1, one unit of flow is augmented along each of the paths *s-d-f-t*, *s-e-f-t* and *s-c-f-t*. After that procedure, the costs c of all edges are updated with respect to the new flow. Along with that, also the reduced costs c^{π} are updated, shown in the residual network in (d). Now again, the shortest distances are computed, the vertex potentials are updated and the admissible network is defined, shown in (e). In the now solved maximum flow problem, one unit of flow is augmented along each of the paths *s-a-c-f-t* and *s-b-c-f-t*, therefore using the edge (c, f) two additional times. The algorithm terminates now because $e(s) = 0$. The original network with the computed flow is shown in Figure 3.6 (a): with the given step cost function, the overall costs are $1+1+5+1+1 = 9$. As we can see in (b), that is not an optimal solution, since the overall costs computed by the SSP are $1 + 1 + 1 + 2 + 2 + 1 = 8$.

This leads to the question why the algorithm did not find the optimal solution. When computing the shortest path distances, the edge costs for sending one more flow were used. Based on those distances, the admissible network was defined and on that, the maximum flow problem was solved. During this computation, the algorithm did not care about the actual costs of augmenting flow along edges, since the cheapest edges have been computed before. While for a linear cost function that leads to optimal flows, it does not for the WCP, as the costs for one more flow were used, disregarding the costs for more flow. Now in the maximum flow problem, the edge (c, f) is used twice at one time, leading to an abrupt increase in the costs that the algorithm did not include in its computations.
In contrast, the modified SSP would not have made that decision that led to a non-optimal solution since there is always just one unit of flow augmented in each step. Thus, the reduced costs are computed again after augmenting the second unit of flow along (c, f),

and the rise in the costs is recognized. The algorithm then uses the cheaper path *c*-*d*-*f* (or *c*-*e*-*f*) instead.
From the possibility of not finding the local optimum within one step it follows that also the global optimum may not be found. Thus, the algorithm is not optimal.

When thinking about other ways to determine the costs c_{ij}, it is quite obvious that other choices end up with exactly the same problem. For example, the costs of sending two more units of flow could be used, but this would lead to wrong results if only one unit was transmitted. Therefore, it does not make sense to determine the costs as a fixed value for each iteration.

Another problem arising due to the step cost function refers to the admissible network. As described in the previous section, it is composed of all zero reduced cost edges. In the original algorithm, $c^{\pi}_{ij} \geq 0$ holds for each edge (i, j) and therefore zero reduced cost edges are the ones to be used in shortest paths. In the WCP, the reduced costs can be negative (as seen in Figure 3.1), making it difficult to define which edges should be contained in the admissible network. If, as in the original algorithm, only the zero reduced cost edges were used, we would obviously ignore cheaper edges, since negative costs are cheaper than zero costs. In case we only took the edges with minimum reduced costs instead, there were possibly no paths at all between vertices (imagine the case where only one edge has negative reduced costs). Therefore the best choice seems to be to use all edges (i, j) with $c^{\pi}_{ij} \leq 0$. However, this could lead to non-optimal results: when solving the maximum flow problem, all edges are treated the same, no matter what their costs are. This leads to correct results in the standard algorithm, because there all used edges have the same reduced costs (namely zero), but not in our case where the costs differ and therefore cheaper edges should be preferred.

Summarized, using the PDA instead of the SSP comes along with some disadvantages: the problem of defining the admissible network, and the problem of taking wrong decisions that did not occur when using the SSP. These decisions lead to results that are equal to those of the SSP in the best case, but can be worse in many cases. Furthermore, as seen in the previous section, the PDA's only advantage over the SSP is the quicker running time while aside of that the algorithms are quite similar. Because of these reasons (the similarity to the SSP, but connected with adding problems while not really offering reasonable advantages), we decide to not adapt the PDA for solving the WCP.

3.2.3 Special Case - One Cable Type Only

In the previous section, we have seen that one problem of the PDA is that it can augment more than one unit of flow along an edge. Once this flow requires a new cable type, the costs can rise abruptly. With only one cable type available, this cannot happen anymore: the costs of the augmented flow, no matter how many units that is, cannot be higher than the costs computed and used by the PDA (i.e. the costs for the next unit of flow) since no new cable type with higher costs can be used. Therefore, the PDA finds the local optimum in each step when only one cable type is used. However, it still does not find the global optimum because it still faces the same problems as the SSP (e.g. for the WCP instance in Figure 3.4 the PDA finds the same non-optimal solution as the SSP).
The other problem of the PDA we encountered was the definition of the admissible network. That problem still occurs with only one cable type, since the reduced costs of edges can still be negative.

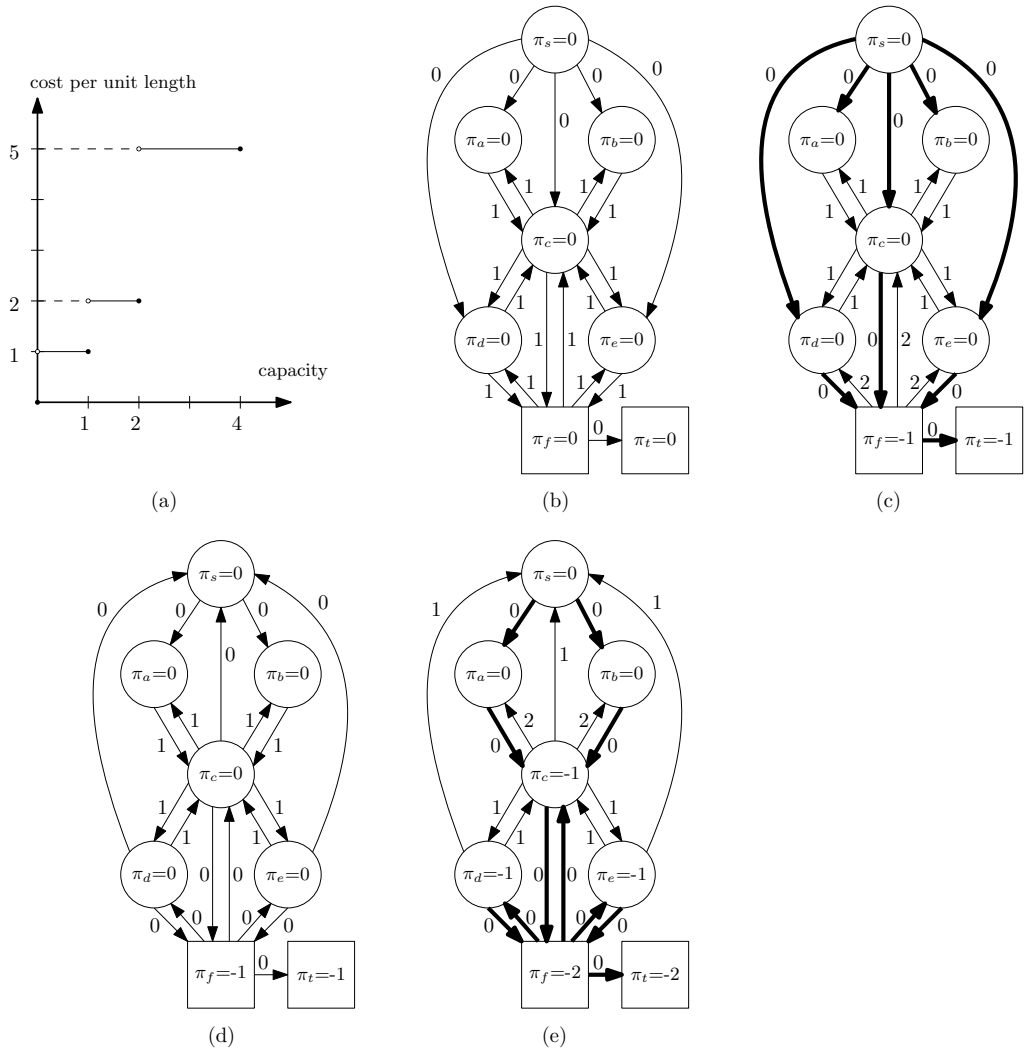

Figure 3.5: Using the PDA for WCP. The value on each edge denotes its reduced costs. Inside each vertex, its potential is displayed. (a) Step cost function; (b) Transformed initial network; (c) Residual network after updating vertex potentials and reduced costs, bold edges belong to the admissible network; (d) Residual network after solving the maximum flow problem und updating the reduced costs; (e) Residual network after updating vertex potentials and reduced costs, bold edges belong to the admissible network.

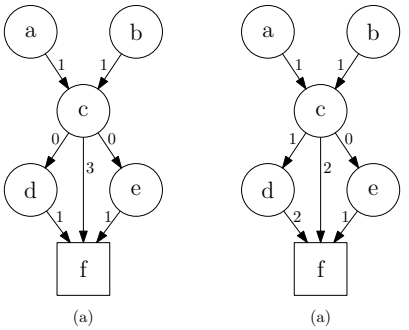

Figure 3.6: Computed solutions. On each edge, the amount of flow along it is shown. (a) Non-optimal flow, computed by the PDA; (b) Optimal flow, computed by the SSP.

3.3 Out-Of-Kilter Algorithm

3.3.1 Describing the Out-Of-Kilter Algorithm

As seen in the last two chapters, the idea of the SSP and the PDA has been to satisfy the reduced cost optimality conditions (Theorem 2.1) and the flow bound constraints (Equation 2.2) in each step. Another approach is to satisfy the mass balance constraints (Equation 2.1) in each step while the reduced cost optimality conditions and the flow bound constraints may be violated. This is the idea of the Out-Of-Kilter Algorithm (OOK) ([AMO93, pp. 326ff.]) that we examine in this chapter.
The OOK is based on the complementary slackness optimality conditions (Theorem 2.2), that are (in a slightly different, equivalent form) restated in the following.

$$\text{If } x_{ij} = 0, \text{ then } c_{ij}^{\pi} \geq 0. \tag{3.3}$$

$$\text{If } 0 < x_{ij} < u_{ij}, \text{ then } c_{ij}^{\pi} = 0. \tag{3.4}$$

$$\text{If } x_{ij} = u_{ij}, \text{ then } c_{ij}^{\pi} \leq 0. \tag{3.5}$$

Throughout the algorithm, every edge of the network is in one of two possible states due to these conditions: it either satisfies the conditions, or it does not; an edge being in the first state is called *in-kilter*, while an edge being in the second state is called *out-of-kilter*. The conditions can be represented by the so-called *kilter diagram* (Figure 3.7): each edge (i, j) defines a point in the diagram based on its flow x_{ij} and its reduced costs c_{ij}^{π}. If that point is located on the bold lines, the edge (i, j) satisfies the conditions (i.e. is in-kilter).

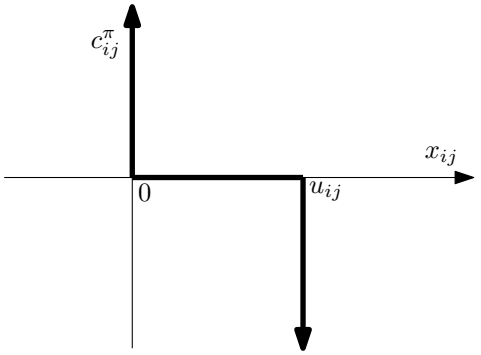

Figure 3.7: Kilter diagram for edge (i, j).

With each edge (i, j), a *kilter number* k_{ij} is associated. That kilter number indicates the required change of flow x_{ij} to transform (i, j) into an in-kilter edge. Depending on c_{ij}^{π}, that is:

$$k_{ij} = \begin{cases} |x_{ij}|, & \text{if } c_{ij}^{\pi} > 0 \\ |u_{ij} - x_{ij}|, & \text{if } c_{ij}^{\pi} < 0 \\ x_{ij} - u_{ij}, & \text{if } c_{ij}^{\pi} = 0 \text{ and } x_{ij} > u_{ij} \\ -x_{ij}, & \text{if } c_{ij}^{\pi} = 0 \text{ and } x_{ij} < 0 \\ 0, & \text{if } c_{ij}^{\pi} = 0 \text{ and } 0 \leq x_{ij} \leq u_{ij}. \end{cases} \tag{3.6}$$

From Equation 3.6 it follows that $k_{ij} \geq 0 \quad \forall (i, j) \in E$. An edge (i, j) with $k_{ij} = 0$ is called an in-kilter edge.

The idea of the algorithm is now to transform all out-of-kilter edges into in-kilter edges. To achieve that, the algorithm decreases in each step the kilter number of one or more edges in the residual network, while never increasing any kilter numbers. Once all kilter numbers equal zero, the flow is both feasible and optimal. The kilter number of an edge (i, j) in the residual network is defined in the following way:

$$k_{ij} = \begin{cases} 0, & \text{if } c_{ij}^{\pi} \geq 0 \\ r_{ij}, & \text{if } c_{ij}^{\pi} < 0. \end{cases} \tag{3.7}$$

This definition of k_{ij} is consistent with the previous definition (Equation 3.6): k_{ij} equals the amount of flow that has to be augmented along (i, j) to satisfy its optimality condition (i.e. the reduced cost optimality condition, Theorem 2.1). If the reduced costs of (i, j) are positive, (i, j) satisfies its optimality condition; if the reduced costs are negative, r_{ij} units of flow have to be augmented along (i, j), so that (i, j) drops out of the residual network and therefore satisfies its optimality condition.

Now we take a closer look at the OOK that is displayed in Algorithm 3.7. After initializing the algorithm with a feasible flow and setting all vertex potentials to zero (lines 1-2), the main loop begins. In each step, one out-of-kilter edge (p, q) is selected (line 4). From the end vertex q of that edge, the shortest distances to all other vertices are computed, leaving out the reverse edge (q, p). For that computation, all negative edge lengths are set to zero (lines 5-6). Particularly, a shortest path P from q to p is stored (line 7). After updating the vertex potentials (line 8), the algorithm performs a check if the selected edge (p, q) is still out-of-kilter (line 9; recap Equation 3.7); if it is, the path P is extended to a cycle by adding edge (p, q) (line 10) and the maximum possible amount of flow (i.e. the minimum residual capacity among the edges on the cycle) is augmented (lines 11-12). That way, the kilter number k_{pq} is decreased. If all edges are in-kilter now, the algorithm terminates; otherwise, the next iteration starts.

Let us now examine the correctness and the termination behaviour of the OOK. The correctness proof is based on the fact that at no time kilter numbers are increased. Only two operations alter the kilter number of an edge: updating the vertex potentials (line 8) and augmenting flow (line 12). The two lemmas [AMO93, Lemma 9.13] and [AMO93, Lemma 9.14] state that neither of those operations increases any kilter number.
Furthermore, in each step the kilter number of at least one out-of-kilter edge (the selected edge in line 4) is decreased by at least 1; either in the update step or in the augmentation step. Therefore, if U denotes the maximum kilter number among all edges, the algorithm terminates after at most $m \cdot U$ steps. In each iteration, solving the shortest path problem dominates the running time. Since that problem is computed on only nonnegative edge weights (line 5), we can for example use Dijkstra's Algorithm, running in $O(n \log n + m)$ time. With that, the OOK has a total running time of $O(m \cdot U \cdot (n \log n + m))$.

3.3.2 Adapting the Out-Of-Kilter Algorithm

Before examining the problems of the Out-Of-Kilter Algorithm on WCP instances, we make some assumptions that we have already made for previous algorithms. To model the capacity of substations, we use the concept of the supersubstation again in the same way we did for the SSP. We consider the costs c_{ij} of an edge (i, j) as the costs that arise when augmenting one more unit of flow than before on that edge. One problem to deal with is the flow augmentation step, since more than one unit of flow can be augmented at one time in the original OOK. As we have seen previously, this can cause problems due to the changing costs of an edge. But, while augmenting the flow by only one in each iteration makes the OOK possibly slower, it does not change the computed results. Therefore, we set $\delta = 1$ in each iteration.

Algorithm 3.7: Out-Of-Kilter Algorithm

Input: Directed Network $G = (V, E)$, capacities u_{ij}, costs c_{ij}
Output: Optimal flow x^*

```
// Initialization
```
$\pi \leftarrow 0$
establish a feasible flow x in G, compute all initial kilter numbers

```
// Main loop
```
while *residual network* $G(x)$ *contains an out-of-kilter edge* **do**
  select an out-of-kilter edge $(p, q) \in G(x)$
  $length(i, j) = \max\{0, c^{\pi}_{ij}\} \quad \forall (i, j) \in G(x)$
  compute shortest path distances $d(\cdot)$ from q to all other vertices in $G(x) - \{(q, p)\}$
  $P \leftarrow$ shortest path from q to p
  $\forall i \in V : \pi(i) = \pi(i) - d(i)$, update all reduced costs
  if $c^{\pi}_{pq} < 0$ **then**
    $W \leftarrow P \cup \{(p, q)\}$
    $\delta \leftarrow \min\{r_{ij} : (i, j) \in W\}$
    augment δ flow along W
    update x, $G(x)$, all reduced costs and all kilter numbers

We now look at the problems that arise when using the OOK to solve the WCP. The first problem is related to the capacities u_{ij}: while in the minimum-cost flow problem each edge has a fixed capacity, we have different capacities for each edge depending on the currently used cable type. In the OOK, the capacities are used for two reasons; the first one is in the augmentation step, where the minimum residual capacity on the cycle is computed. Since we already decided to augment only one unit of flow per step, we avoided any problems here (since each edge in the residual network has a capacity of at least one). The second use is when computing the kilter number of an edge (Equation 3.7). There we have two possibilities: using the capacity of the currently used cable type, or always using the maximum capacity among the cable types. The second possibility seems to be the better choice, as the first one would incorrectly reflect the amount of flow that could be augmented along an edge.

The next problem we encounter is related to the fundamental idea of the OOK. As seen in the previous section, the algorithm is based on the complementary slackness conditions, since these conditions state whether an edge is in-kilter or out-of-kilter and thus has to be modified. These conditions, on the other hand, are based on the fact that for each edge (i, j) the following skew-symmetry (Equation 3.8) holds:

$$c^{\pi}_{ij} = -c^{\pi}_{ji}. \tag{3.8}$$

We briefly show how the complementary slackness condition in Equation 3.4 is based on Equation 3.8. From the reduced cost optimality conditions (Theorem 2.1) we know that for an optimal solution, each c^{π}_{ij} has to be nonnegative. If the flow x_{ij} of an edge (i, j) is now greater than zero and less than the maximum capacity, the residual network also contains its reversal edge (j, i). Since $c^{\pi}_{ij} = -c^{\pi}_{ji}$, the reduced costs of both (i, j) and (j, i) can be nonnegative if and only if $c^{\pi}_{ij} = -c^{\pi}_{ji} = 0$, which is exactly condition 3.4. Therefore, 3.8 is a sufficient condition for the complementary slackness condition 3.4. While that skew-symmetry is always satisfied for a linear cost function, it sometimes is not for step cost functions, which we use in the WCP. Figure 3.8 shows a simple example where Equation 3.8 is not satisfied and therefore an edge is wrongly classified as being out-of-kilter. The

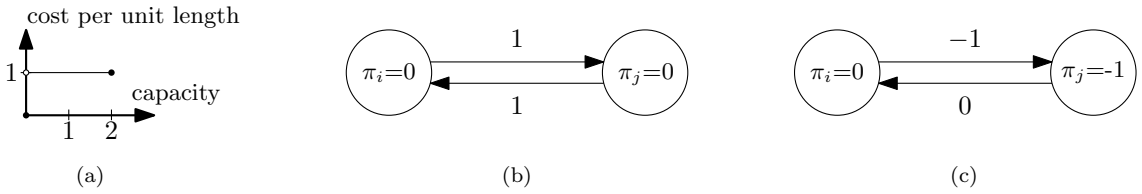

Figure 3.8: Complementary slackness conditions in WCP. The value on each edge denotes its reduced costs. Inside each vertex, its potential is displayed. (a) Step cost function; (b) Initial network; (c) Network after the first iteration and augmenting one unit of flow along (i, j).

network consists only of the two vertices i and j, thus the only way to augment flow from i to j is along edge (i, j). In (c) we see the network after updating the vertex potentials and increasing the flow on edge (i, j) by one: now $c^{\pi}_{ij} = -1$ and $c^{\pi}_{ji} = 0$, so $c^{\pi}_{ij} \neq -c^{\pi}_{ji}$; therefore Equation 3.8 is not satisfied. Although both edges (i, j) and (j, i) are in an optimal state now (recall that there was no other way of augmenting flow), Equation 3.4 defines (i, j) as being out-of-kilter.
Without that skew-symmetry, the complementary slackness conditions do not provide the intended informative value about the optimality of a flow anymore. Hence, the previous definition of in-kilter and out-of-kilter edges does not make sense anymore, since that definition was based on the complementary slackness conditions.

Now, what does that mean for the question if we can use the OOK to solve the WCP? Obviously, the optimality conditions that the OOK is based on, do not hold for WCP networks. Therefore, the algorithm could consider optimal edges as non-optimal and vice versa. While one could now assume that only the optimality of solutions computed with the OOK is not given anymore, the arising problems are even greater. Recall that the algorithm aims to transform all edges into in-kilter edges and that the definitions of in-kilter and out-of-kilter edges make no sense anymore. As we have seen before, the proof of correctness and termination is based on the lemmas [AMO93, Lemma 9.13] and [AMO93, Lemma 9.14]. These lemmas state that kilter numbers never increase during the algorithm. With Figure 3.9 we now show an example where (due to the step cost function) kilter numbers increase when using the OOK to solve the WCP. In (b) the WCP network and a feasible initial flow is shown: vertex c is the only substation, vertices a, b, d and e are turbines, and vertex t is the supersubstation. All edges have unit length. In the graphs shown in (c), (d) and (e), the turbines d, e and the supersubstation are omitted since they are not relevant in any step for the problem. In (c), the initial reduced costs are displayed. In the first step, the edge (b, a) is selected as an out-of-kilter edge, since its kilter number is greater than zero due to Equation 3.7. The algorithm now computes the shortest path distances from vertex a to all other vertices in $G(x) - \{(a, b)\}$. Recall that for this computation, all negative edge lengths are set to zero (Algorithm 3.7, line 5). In (d) we see the reduced costs after updating the vertex potentials with these distances. The shortest path from vertex a to vertex b has been computed as a-c-b. Since now $c^{\pi}_{ba} = -1 < 0$, one unit of flow is augmented along the cycle a-c-b-a, and the first iteration ends. The updated reduced costs are shown in (e). Now we take a look at edge (a, c): at the beginning of the step, $c^{\pi}_{ac} = 2$. Therefore, its kilter number has been 0 according to Equation 3.7, so the edge was in-kilter. At the end of the step, $c^{\pi}_{ac} = -2$ and $x_{ac} = 1$, thus $k_{ac} = r_{ac} = u_{ac} - x_{ac} = 2 - 1 = 1$. We observe that the step not only increased the kilter number of edge (a, c), but even transformed it from an in-kilter edge into an out-of-kilter edge. The reason why that happened is that after augmenting flow along (a, c), its costs c_{ac} decreased to zero, since the now used cable type has an unsaturated capacity of 2.
Recall that we stated before the two possibilities of defining the capacity of an edge. In

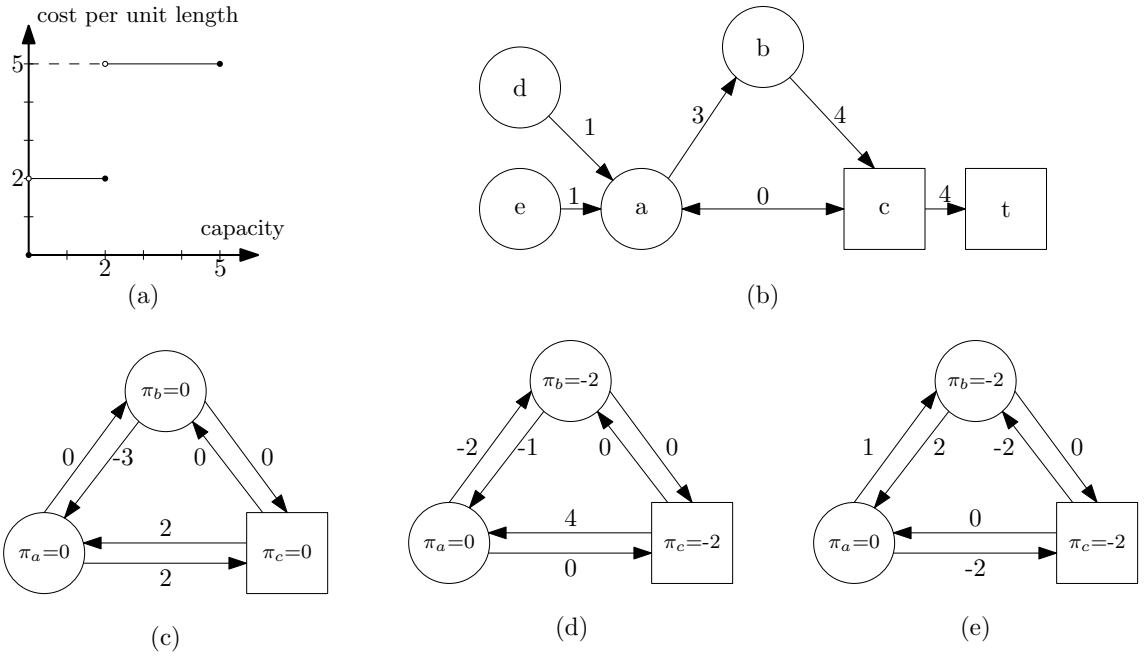

Figure 3.9: Increasing kilter numbers when using the OOK for the WCP. All edges have unit length. In (c)-(e), each edge is labeled with its reduced costs and each vertex is labeled with its potential. (a) Step cost function; (b) WCP network with an initial feasible flow displayed on the edges; (c) Initial reduced costs and vertex potentials; (d) Updated reduced costs and vertex potentials after computing the shortest distances; (e) Reduced costs after augmenting one unit of flow along *a*-*c*-*b*-*a*.

that example, we used the residual capacity of the currently used cable type. If instead we use the residual capacity of the largest available cable type, the kilter number of (a, c) is computed as $k_{ac} = r_{ac} = u_{ac} - x_{ac} = 5 - 1 = 4$, and therefore (a, c) is classified as an out-of-kilter edge too.

So we have seen that the correctness proof of the standard OOK does not hold anymore for step cost functions, as kilter numbers may increase during the algorithm. While that theoretical analysis does not give any information about the actual termination behaviour, preliminary experiments have shown that infinite loops occur.

Summarized, we meet a prohibitive problem when trying to solve the WCP with the OOK. The optimality conditions used by the OOK to define whether an edge is either in an optimal or non-optimal state do not work with a step cost function. Because of that, fundamental concepts as the correctness and termination proof do not hold for WCP instances, eventually leading to infinite computations. Regarding the fact that the whole underlying concept of the OOK does not work for the WCP and since we see no straightforward way to adapt the optimality conditions to become meaningful for step cost functions, the OOK will not be adapted further.

3.3.3 Special Case - One Cable Type Only

In the previous section, we encountered two problems, one concerning the definition of residual capacities of edges, and the other one with respect to the complementary slackness optimality conditions. If only one cable type is available, the first problem does not occur anymore, since in that case each edge has a fixed capacity. Therefore, defining the residual capacity of an edge works similar to the standard OOK.

The more serious problem with the definition of in- and out-of-kilter edges still occurs: as we have seen before, Figure 3.8 uses only one cable type and still an edge was wrongly classified. Preliminary experiments have shown that also with only one cable type, kilter numbers can increase and infinite loops occur.

3.4 Relaxation Algorithm

3.4.1 Describing the Relaxation Algorithm

In this chapter, we take a closer look at the Relaxation Algorithm (RLX) as it is described in [AMO93, pp. 332ff.].
The idea of the RLX is to use the *Lagrangian relaxation* technique ([AMO93, pp. 605ff.]) that provides a so-called relaxed problem, which is directly related to the minimum cost flow problem as we see later. The RLX relaxes the mass balance constraints; that means, for each vertex i, its mass balance constraint is multiplied by a scalar $\pi(i)$, referred to as the vertex potential, and the resulting product is subtracted from the objective function (which is the overall flow costs):

$$w(\pi) = \min_x \sum_{(i,j)\in E} c_{ij}x_{ij} + \sum_{i\in V} \pi(i) \left(- \sum_{j\in V:(i,j)\in E} x_{ij} + \sum_{j\in V:(j,i)\in E} x_{ji} + b(i) \right) \tag{3.9}$$

subject to

$$0 \leq x_{ij} \leq u_{ij} \quad \forall (i,j) \in E. \tag{3.10}$$

For a given set of vertex potentials π, that relaxed problem is denoted as $LR(\pi)$, and its objective function is denoted as $w(\pi)$. We can restate Equation 3.9 in the following equivalent ways (recap that $b(i)$ denotes the supply/demand and $e(i)$ denotes the imbalance (Equation 2.4) of a vertex i):

$$w(\pi) = \min_x \sum_{(i,j)\in E} c_{ij}x_{ij} + \sum_{i\in V} \pi(i)e(i), \tag{3.11}$$

$$w(\pi) = \min_x \sum_{(i,j)\in E} c^{\pi}_{ij}x_{ij} + \sum_{i\in V} \pi(i)b(i). \tag{3.12}$$

With the formulation in Equation 3.12, we can obtain an optimal solution x of $LR(\pi)$ for a given set of vertex potentials π in the following way:

$$\text{If } c^{\pi}_{ij} > 0, \text{ then } x_{ij} = 0. \tag{3.13}$$

$$\text{If } c^{\pi}_{ij} < 0, \text{ then } x_{ij} = u_{ij}. \tag{3.14}$$

$$\text{If } c^{\pi}_{ij} = 0, \text{ then } x \text{ can be set to any value between 0 and } u_{ij}. \tag{3.15}$$

That solution is a pseudoflow (section 2.1) for the minimum cost flow problem which satisfies the reduced cost optimality conditions (Theorem 2.1). Therefore, we can state the following property ([AMO93, Prop. 9.15]):

Property 3. *If a pseudoflow x of the minimum cost flow problem satisfies the reduced costs optimality conditions for some π, then x is an optimal solution of $LR(\pi)$.*

The next lemma (adapted from [AMO93, Lemma 9.16]) now shows how the relaxed problem is related to the original problem:

Lemma 3.1. *Given an optimal flow x^* for the minimum cost flow problem with total costs $z(x^*)$:*
(a) For any vertex potentials $\pi : w(\pi) \leq z(x^*)$.
(b) For some choice of vertex potentials $\pi^* : w(\pi^*) = z(x^*)$.

Thus, if for some flow x and some vertex potentials π the equation $w(\pi) = z(x)$ is true, then x is an optimal solution of the minimum cost flow problem. This lemma is used later again when it comes to the proof of termination of the RLX.

At all times, the RLX maintains a pair (x, π) where the pseudoflow x is an optimal solution of $LR(\pi)$ and therefore (x, π) satisfies the reduced cost optimality conditions. In each step the algorithm modifies the pseudoflow x to x' in a way that x' is also an optimal solution of $LR(\pi')$. In addition, it either decreases the excess of at least one vertex, or modifies π to π' in a way that $w(\pi') > w(\pi)$. If possible, the algorithm chooses the second option. Now we take a look at the RLX (Algorithm 3.8) in more detail.
The algorithm starts with zero flow and zero vertex potentials and then performs *major iterations*. In each major iteration, an excess vertex s is selected (line 3). Beginning with that vertex, a tree S is built in a way that all tree vertices have nonnegative imbalances and all tree edges have zero reduced costs. Adding a vertex to the tree is called a *minor iteration*. Before we see how the tree grows and when a major iteration ends, we introduce some notation. The set of vertices that do not belong to the tree is denoted as $\overline{S}$. With that, we define the forward cut $(S, \overline{S}) = \{(i, j) \in E : i \in S \text{ and } j \in \overline{S}\}$ and the backward cut $(\overline{S}, S) = \{(i, j) \in E : i \in \overline{S} \text{ and } j \in S\}$. Furthermore, we define the overall tree imbalance $e(S) = \sum_{i \in S} e(i)$ and the overall residual capacity of zero reduced costs cut edges $r(\pi, S) = \sum_{(i,j) \in (S,\overline{S}) \text{ and } c_{ij}^{\pi} = 0} r_{ij}$.
Whenever a vertex is added to the tree (i.e. at the beginning (line 4) or in a minor iteration (line 11)), the algorithm checks if $e(S) > r(\pi, S)$ (line 5, line 12/13). If that is true, the procedure *adjust-potential* 3.9 is performed. In the first step of that procedure, all zero reduced costs edges in $(S, \overline{S})$ are saturated (line 1) and since $e(S) > r(\pi, S)$, the remaining imbalance of the tree is still positive. Note that this operation does not change $w(\pi)$ since the flow change affects only zero reduced cost edges. Now all edges in $(S, \overline{S})$ have strictly positive reduced costs because the zero reduced cost edges dropped out of the residual network. Among the remaining forward cut edges, the lowest reduced costs are computed (denoted as α, line 2) and all vertex potentials in S are increased by that value (line 3). That way, $w(\pi)$ is increased by $(e(S) - r(\pi, S))\alpha$ units while preserving the reduced cost optimality conditions. The current major iteration is finished.
If the check $e(S) > r(\pi, S)$ is false, the algorithm tries to grow the tree. A zero reduced cost edge (i, j) from the forward cut is selected (line 8; note that at least one such edge exists, since $0 < e(S) \leq r(\pi, S)$). If the imbalance of j is nonnegative, j is added to S and its parent is stored (lines 10-11). If still $0 < e(S) \leq r(\pi, S)$, the next minor iteration begins, otherwise the *adjust-potential* procedure is performed (line 14). If the imbalance of j is negative, the procedure *adjust-flow* 3.10 is executed. Using the parent pointers of the tree, the directed path from the start vertex s (which has strictly positive imbalance) to j is computed (line 1) and the maximum possible amount of flow (line 2) is augmented along that path (line 3). This operation decreases the excess of s while it does not change $w(\pi)$ since it takes place only on zero reduced cost edges. The current major iteration is finished.

The algorithm terminates when all vertices have an imbalance of zero. Since the pair (x, π) maintained throughout the algorithm satisfies the reduced cost optimality conditions at all times, the resulting flow is a minimum cost flow. We now take a look at the proof of termination and the running time.
As we have seen, each major iteration ends by performing either the adjust-potential or the adjust-flow procedure. While the first one strictly increases $w(\pi)$, but might also increase the overall excess, the second procedure strictly decreases the overall excess and does not change $w(\pi)$. We now show that the algorithm still terminates in a finite number of iterations. In the following, U denotes the largest magnitude among all vertex imbalances and edge capacities; C denotes the largest cost among all edges.

Algorithm 3.8: Relaxation Algorithm

Input: Directed Network $G = (V, E)$, capacities u_{ij}, costs c_{ij}
Output: Optimal flow x^*

```
// Initialization
x ← 0
π ← 0
// Main loop
while G(x) contains a vertex s with e(s) > 0 do
    S ← {s}
    if e(S) > r(π, S) then
        adjust-potential(G, S)
    repeat
        select an edge (i, j) ∈ (S, S̄) with c^π_ij = 0
        if e(j) ≥ 0 then
            parent(j) ← i
            add j to S
    until e(j) < 0 or e(S) > r(π, S)
    if e(S) > r(π, S) then
        adjust-potential(G, S)
    else
        adjust-flow(G, S, s, (i,j))
```

Algorithm 3.9: Adjust-Potential Procedure

Input: Directed Network $G = (V, E)$, Tree S
Output: Updated flow and vertex potentials

```
∀(i, j) ∈ (S, S̄) with c^π_ij = 0: augment r_ij units of flow along (i, j)
α ← min{c^π_ij : (i, j) ∈ (S, S̄) and r_ij > 0}
∀i ∈ S : π(i) ← π(i) + α
```

Algorithm 3.10: Adjust-Flow Procedure

Input: Directed Network $G = (V, E)$, Tree S, start vertex s, edge (i, j)
Output: Updated flow

```
P ← directed path from vertex s to vertex j //Trace back parents
δ ← min[e(s), −e(j), min(r_ij : (i, j) ∈ P)]
augment δ units of flow along P
```

Using the upper bound mCU on the total costs of a feasible flow, Lemma 3.1(a) yields mCU as the maximum possible value of $w(\pi)$. Since the algorithm starts with $w(\pi) = 0$ and $w(\pi)$ is never decreased, and each call of adjust-potential increases $w(\pi)$ by at least 1, the total number of adjust-potential calls is in $O(mCU)$. Between two of those executions, the algorithm may perform several adjust-flow calls. After each adjust-potential, the overall excess is at most $O((m + n)U) = O(mU)$. Each execution of adjust-flow decreases the overall excess by at least 1. Therefore, the number of adjust-flow executions between two adjust-potential executions is $O(mU)$. Hence, altogether the number of major iterations is $O(m^2CU^2)$. In each major iteration, at most n minor iterations are performed, since at most n vertices can be added to the tree. With that, the total number of minor iterations

of the RLX is in $O(m^2nCU^2)$.
While that worst-case running time is much worse than those of the previous algorithms, the RLX has proven to be very efficient in practice; on most classes of networks, it outperforms the previous algorithms, and even competes with the network simplex algorithm ([AMO93, p. 332]).

3.4.2 Adapting the Relaxation Algorithm

Now we examine how the Relaxation Algorithm can be adapted to work on WCP instances. Just like in the previous algorithms, we use the concept of the supersubstation to model the capacities of the substations. Again, the supersubstation collects the demand of all substations and is connected with them via zero cost edges.
Also similar to the previous algorithms, the costs c_{ij} of an edge (i, j) are the current costs of sending one more unit of flow along (i, j). As seen in the last section, the RLX can (and often does) augment more than one unit of flow along an edge at one time. Since the costs c_{ij} are not linear in the WCP, they can decrease or increase when augmenting flow, leading to non-optimal decisions because the algorithm bases its computations on a fixed c_{ij}. Therefore, an approach to achieve better solutions is to augment only one unit of flow per edge and iteration. To apply this decision, in line 1 of the adjust-potential procedure (Algorithm 3.9) r_{ij} is replaced with 1, and in line 2 of the adjust-flow procedure (Algorithm 3.10) δ is set to 1. While this may slow down the algorithm, it does not affect the termination behaviour and the optimality of the solution of the standard algorithm.
Another problem is the definition of the residual capacity r_{ij} of an edge (i, j). While the standard RLX deals with networks with fixed capacities for each edge, the capacity of an edge in WCP is determined by the cable types. Thus, we have two options to determine the capacity u_{ij} and therefore r_{ij}. First, we could use the residual capacity of the currently used cable type, i.e. the amount of flow that can be augmented along (i, j) before a new cable type is required. If the currently used cable has no capacity left, we use the residual capacity of the next larger cable type, since that matches with the idea of the used costs (remember that we use the costs for augmenting one more unit of flow). The second option is to use the residual capacity of the largest available cable type. While that matches with the idea of the standard RLX, it ignores the fact that the WCP has the choice of not only one, but multiple cable types. Thus, both options seem to be feasible and have both advantages and disadvantages.
In the original algorithm, no negative edge weights occur during runtime. In contrast (as we have seen before in Figure 3.1), this is not true for the WCP because of the step cost function. In the adjust-potential procedure of the original algorithm, the α computed as the lowest reduced costs among edges in $(S, \overline{S})$ is strictly positive and therefore the potentials of the vertices in S are increased. With negative reduced costs, α can be negative, as we see in Figure 3.10: in (d) the costs c_{ij} decrease from 1 to 0 and therefore the reduced costs c^{π}_{ij} decrease from 0 to -1. The way we can handle those negative costs is the same as the standard algorithm does: we compute α as $\alpha = \min\{c^{\pi}_{ij} : (i, j) \in (S, \overline{S}) \text{ and } r_{ij} > 0\}$. If that value is negative, the potentials are decreased (as shown in (e)). That way, all edges in $(S, \overline{S})$ have nonnegative costs and at least one edge has zero reduced costs, just like in the standard algorithm.

Applying these adaptions, preliminary experiments have shown that the RLX works on some instances only, while never terminating on others. We now examine why these infinite computations occur. As we have seen before, the idea and the proof of termination of the RLX is based on the relaxed problem $LR(\pi)$ and the fact that its objective function $w(\pi)$ (Equation 3.9) is never decreased. That property is no longer given on WCP instances; both the adjust-flow and the adjust-potential procedure can now decrease $w(\pi)$. In Figure 3.10 (d) we see that the adjust-flow call decreases c^{π}_{ij} and increases x_{ij}. Thus, the first term of

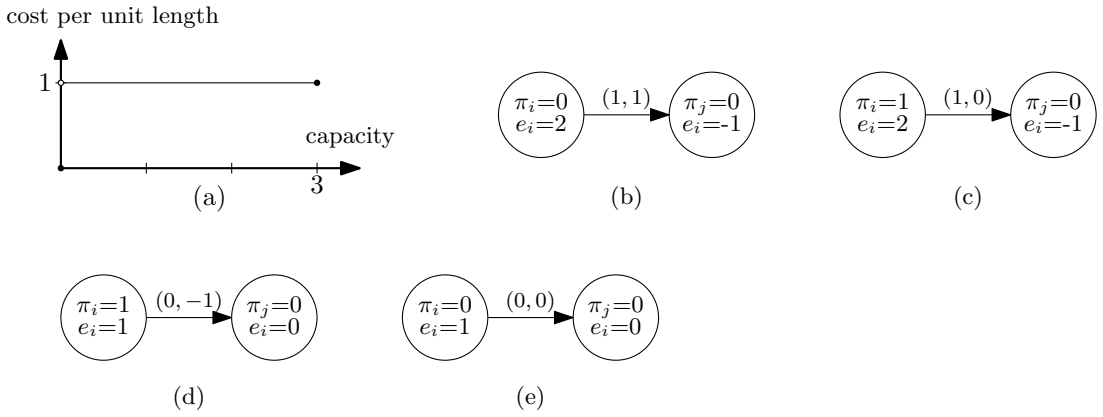

Figure 3.10: Negative edge weights and negative values for α. The pair on each edge denotes its costs and its reduced costs. (a) The step cost function; (b) Initial situation. Until here, $x_{ij} = 0$; (c) Updated potentials and reduced costs after adjust-potential with $\alpha = 1$; (d) Updated imbalances, costs and reduced costs after adjust-flow; (e) Updated potentials and reduced costs after adjust-potential with $\alpha = -1$.

$w(\pi)$ (formulation in Equation 3.12) decreases by 1 while the second term does not change. In (e) we see that the adjust-potential call decreases $\pi(i)$, therefore decreases the second term of $w(\pi)$ (formulation in Equation 3.11) by 1 while the first term does not change. In both cases, the total value of $w(\pi)$ strictly decreases.
So the guarantee of termination can no longer be shown with the previous argumentation, and thus we have no information about the actual termination behaviour in theory. In the following we show that infinite loops occur in practice. In Figure 3.11 (b), the initial WCP instance is shown. The supersubstation has been omitted in the representation. With the exception of (f, g), which has a length of 20, all edges have unit length. The general concept of the infinite loop is to get into a state where flow is sent back and forth between the two vertices e and f infinite times. The first procedure to reach that state is to send one additional unit of flow to vertex e. To do so, the vertices a, b, c and d are used: they grow a tree with an excess high enough that it exceeds the residual capacity of edge (d, e). After that, both units of flow from e are sent to f. From there on, the vertices f and e send one unit of flow back and forth infinite times. Again, the left vertices are used to "collect" enough excess to exceed the residual capacity of (e, f).
We now look at the steps in more detail. In the first four iterations, the algorithm always starts with $S = \{a\}$ and ends with an adjust-potential call. In the i-th iteration, a tree consisting of i vertices (going from a to d) is grown, until the forward cut $(S, \overline{S})$ contains no zero reduced cost edge. At this point in each iteration, the only edge in $(S, \overline{S})$ has reduced costs of 5, so adjust-potential is performed with $\alpha = 5$ and thus the potentials of all i vertices in S are increased by 5. This makes the edge in $(S, \overline{S})$ a zero reduced cost edge, since its reduced costs are decreased by 5. The result after these iterations is shown in (c). The next step starts again with $S = \{a\}$ and adds b, c, and d to S. Now $e(S) = 4 > 3 = r(\pi, S)$, therefore adjust-potential is performed and one unit of flow is augmented along (d, e). Since now $c_{de} = 0$ and $c^{\pi}_{de} = -5$, α is -5 and therefore the potentials of the vertices in S are decreased by 5 (result is shown in (d)). The next step starts with $S = \{b\}$ and adds c, d and e to S and performs an adjust-potential call with $\alpha = c^{\pi}_{ef} = 5$ (result shown in (e)). The next iteration starts the same way, but now after adding vertex e to S, it holds that $e(S) = 4 > 3 = r(\pi, S)$ and (e, f) is a zero reduced cost edge, thus one unit of flow is augmented along it. Then α is computed as $c^{\pi}_{ef} = -5$ and the potentials of the vertices in S are updated with that value. The resulting residual network is shown in (f),

and that state is the beginning of the endless loop.
The next iteration is similar to the previous one, with the only difference that this time $\alpha = c^{\pi}_{ef} = 0$. The residual networks after this step is shown in (g).
The next iteration starts with $S = \{f\}$. Since $e(S) = 3 > 1 = r(\pi, S)$ and $c^{\pi}_{fe} = 0$, one unit of flow is augmented along (f, e). Now $c_{fe} = c^{\pi}_{fe} = -5$, therefore $\alpha = -5$ and the potential of f is decreased by 5 (result shown in (h)). In the next step, the algorithm again chooses $S = \{b\}$, adds c, d and e to S and calls adjust-potential with $\alpha = c^{\pi}_{ef} = -5$. The residual network after that operation is displayed in (i) and shows exactly the same state we already had before (residual network shown in (f)), with the only difference that the vertex potentials of some vertices have been decreased. This loop is now performed again infinite times and the algorithm never terminates.

We stated earlier that there are two options of determining the residual capacity of an edge. In the prior example, we used the first option (i.e. using the residual capacity of the currently used cable type). Using the second option does not prevent the infinite loops, since negative edge weights still occur. Another approach is to dismiss the idea of augmenting only one unit of flow per iteration, which we used to obtain better results. Recall that the original adjust-potential procedure saturates edges. Therefore they drop out of the residual network and cannot be negative. Still, it does not fix the problem since the original adjust-flow procedure augments $\delta = \min[e(s), -e(j), min(r_{ij} : (i, j) \in P)]$ units of flow and therefore does not always saturate edges. Thus, the problem of negative reduced costs still occurs and can cause the algorithm to run into an infinite loop.

Still, as mentioned before, the algorithm works on some WCP instances and computes feasible flows. Although it is not sufficient to work on some instances, we briefly talk about the optimality of the algorithm. Just like the modified SSP, the modified RLX is not optimal: on the instance shown in Figure 3.12 (b), it computes the flow shown in (c). That flow has a total cost of 40, while the optimal solution has a cost of 32 (see Figure 3.3 (f)). The problem here is that edge (a, d) has very high costs after augmenting the first unit of flow along it. This caused the algorithm not to use that edge again, not considering that using it two more times would be cheaper than the other options.

Summarized, we meet a similar problem as in the OOK: the fundamental idea of the Relaxation Algorithm that guarantees both optimality and a termination in a finite number of steps does not work for WCP instances. The step cost function breaks the termination behaviour and may cause infinite loops. This happens in all our tried variations of computing the residual capacities of edges and the amount of flow augmented in a single step. Furthermore, even on the instances where the algorithm provides a feasible solution, it is not optimal. Mainly due to the problem of not terminating in general, the RLX will not be adapted further.

3.4.3 Special Case - One Cable Type Only

In the previous section, we encountered two problems: how can we define residual capacities of edges and the possible decrease of the objective value $w(\pi)$ that leads to infinite loops. The first of those problems does not arise in case only one cable type is available: defining the residual capacity of an edge can now be done in the same way as in the standard RLX, since each edge has a fixed capacity now.
Though, using only one cable type does not prevent $w(\pi)$ from being decreased: in the previous example (Figure 3.10), where we have shown that $w(\pi)$ can decrease, only one cable type was used. Therefore, the termination behaviour of the RLX is still unclear, and preliminary experiments have shown that infinite loops still occur with just one cable type available.

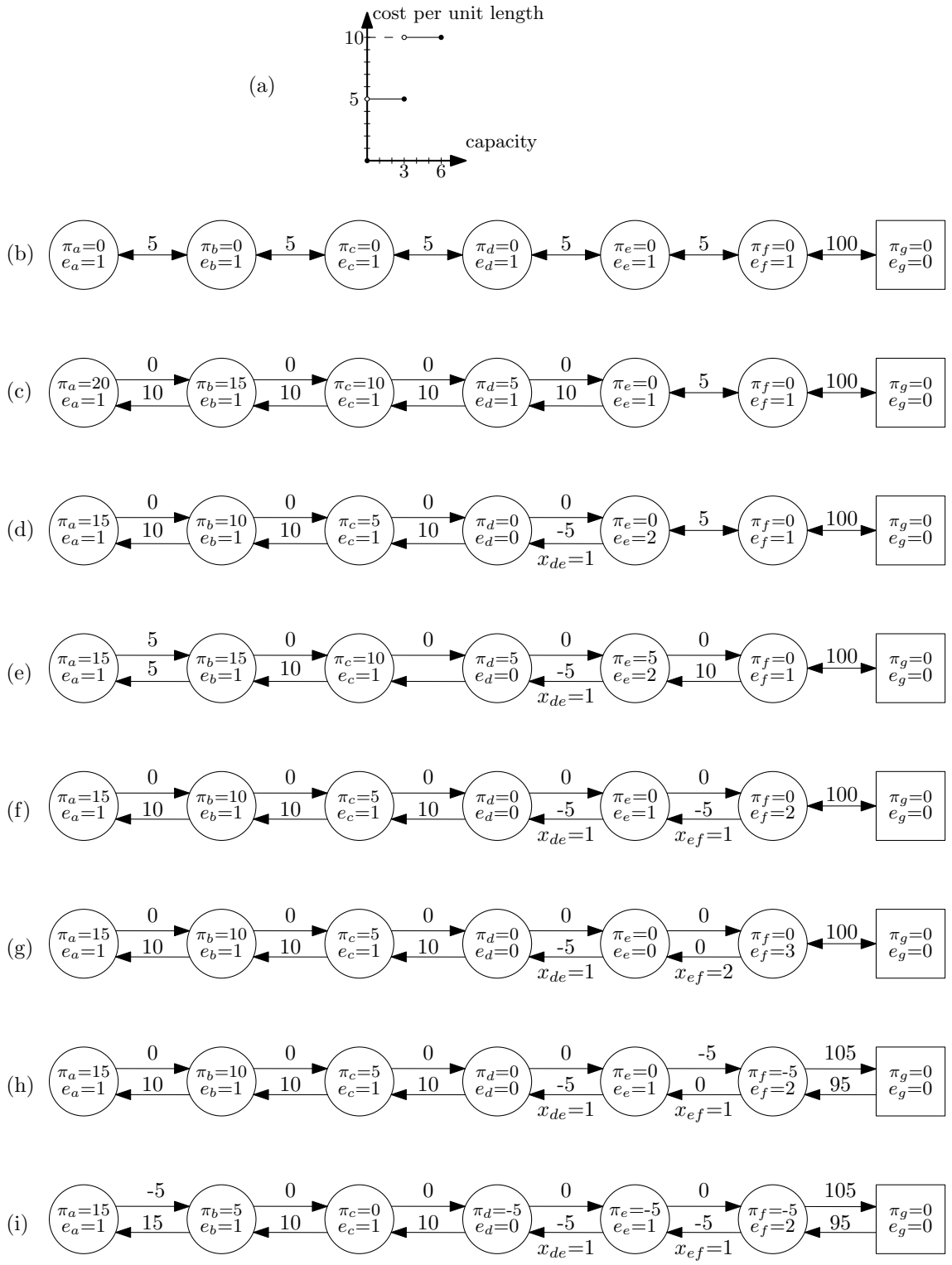

Figure 3.11: Infinite loop with the RLX. The value on each edge denotes its reduced costs. Non-zero edge flows are labeled below the edge. Inside each vertex, its potential and imbalance is displayed. (a) Step cost function; (b) Initial network; (c) Residual network after four major iterations, each starting with vertex a; (d) Residual network after augmenting one unit of flow along d-e; (e) Residual network after one major iteration, starting with b; (f) Residual network after augmenting one unit of flow along e-f; (g) Residual network after augmenting one unit of flow along e-f; (h) Residual network after augmenting one unit of flow along f-e; (i) Residual network after one major iteration, starting with b.

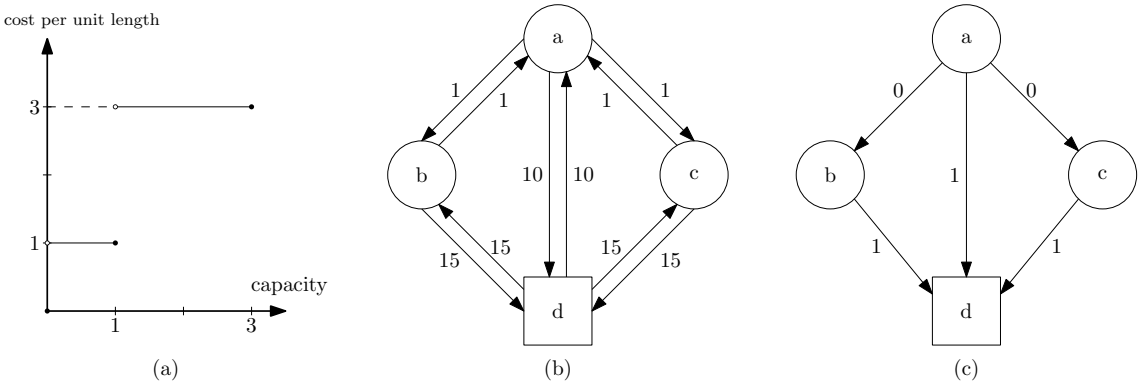

Figure 3.12: Non-optimal solution with the RLX. (a) Step cost function; (b) Initial network, each edge is labeled with its costs; (c) Computed solution, each edge is labeled with the amount of flow along it.

4. Experimental Evaluation

In the previous sections, we developed one working algorithm in four variants which solve the WCP. In this chapter, we first compare these variants among each other to find the best one in terms of running time and solution quality. This variant is then compared to a negative cycle canceling (NCC) approach [GUW+19] and a MILP variation, again with respect to quality of the found solutions as well as the required running times.
For these evaluations we use benchmark sets for wind farms from [LRWW17], with each set having different wind farm sizes and characteristics. Set $\mathcal{N}_1$ contains small wind farms with one substation and 10-79 turbines, while $\mathcal{N}_2$ contains wind farms with multiple substations and 20-79 turbines. Larger wind farms are found in $\mathcal{N}_3$ and $\mathcal{N}_4$, having multiple substations and 80-180 (200-499, respectively) turbines. $\mathcal{N}_5$ consists of complete graphs with a size of 80-180 turbines and multiple substations.
The SSP code is written in C++14 and compiled with GCC 8.2.1. A 64-bit architecture with four 12-core CPUs of AMD clocked at 2.1 GHz with 256 GB RAM running OpenSUSE Leap 15.1 was used to run the simulations. All simulations were computed in single-threaded mode. That setup is nearly identical to the setup used to run the NCC and MILP experiments ([GUW+19]) to ensure comparability of all algorithms.
In the following evaluations, we often look at ratios of the two compared algorithms. When doing so, we always divide the value of the first mentioned algorithm by the value of the second one.

4.1 Comparing Variants of the SSP

We briefly recall which variants we developed in the previous chapter. We have two strategies of finding shortest paths (Dijkstra's Algorithm and an adapted Bellman-Ford Algorithm) and two strategies of choosing the costs of an edge (using vertex potentials and reduced costs or using the normal costs). These strategy options provide us four possible variants, which we referred to as DIJNOR, DIJPOT, BELNOR and BELPOT.
To find the best of these variants, we evaluate them in two steps. First, we compare both Dijkstra variants among each other, and separately both Bellman-Ford variants. In the second step, we compare the better Dijkstra variant to the better Bellman-Ford variant to find the overall best SSP variant.
For these evaluations we randomly select 200 instances per benchmark set, so they are independently chosen from the selected instances in [GUW+19].

Table 4.1: Minimum, average and maximum of running times in milliseconds of both compared variants. For each column, the minimum (i.e. the best) value is marked in green. The last row displays the minimum, average and maximum time ratios.

Strategy Combination	$\mathcal{N}_1$			$\mathcal{N}_2$			$\mathcal{N}_3$			$\mathcal{N}_4$			$\mathcal{N}_5$		
	min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
DIJNOR	0.13	2.47	12.1	0.58	3.98	12.8	11.9	28.3	88.6	122	573	1.4k	167	872	4.8k
DIJPOT	0.13	2.56	12.8	0.57	4.03	12.6	12.0	29.4	95.9	131	666	1.7k	125	609	3.1k
Time ratio	0.88	0.99	1.21	0.91	0.99	1.17	0.88	0.97	1.07	0.79	0.88	1.03	1.22	1.38	1.71

4.1.1 Comparing DIJNOR to DIJPOT

We begin the evaluation by comparing the running times of both strategies. The first two rows of Table 4.1 show the minimum, average and maximum running times for each benchmark set. We also look at the time ratio of both algorithms, i.e. we divide the runnning time of DIJNOR by the running time of DIJPOT; for each set, the minimum, average and maximum time ratios are displayed in the last row of Table 4.1.
Our first observation is that both variants are very fast, with maximum running times of 4.8 seconds (DIJNOR) and 3.1 seconds (DIJPOT), both on $\mathcal{N}_5$. On the benchmarks sets $\mathcal{N}_1$, $\mathcal{N}_2$ and $\mathcal{N}_3$, both variants have quite similar running times with average time ratios of 0.99, 0.99 and 0.97. DIJNOR is slightly faster on these sets: while it outruns DIJPOT on 58 % of the instances on $\mathcal{N}_1$ and $\mathcal{N}_2$, it is faster on 82.5 % on $\mathcal{N}_3$. On large graphs (set $\mathcal{N}_4$), we observe significant differences: DIJNOR is faster on all but two instances, having an average ratio of 0.88. In contrast, DIJPOT is faster on complete graphs ($\mathcal{N}_5$), as it outruns DIJNOR on each instance. The ratios range from 1.22 to 1.71, showing that DIJPOT is much faster on these graphs. It remains unclear why both strategies perform so differently on these two sets.
Overall, both variants show similar running times, with DIJNOR being faster on 59.4 % of all instances, but with an overall average ratio of 1.044 that indicates a small advance for DIJPOT, primary coming from set $\mathcal{N}_5$. Since both variants are very fast and differ only slightly in running times, we base our decision which variant is better solely on the quality of their solutions.

To evaluate the quality of the solutions found by both strategies, we compute for each instance the relative cost ratio, i.e. we divide the objective value found by DIJNOR by the objective value found by DIJPOT. For each benchmark set, the ratios of all instances are sorted in ascending order and then plotted in Figure 4.1. The first observation is that DIJNOR finds way better solutions on nearly all instances. DIJPOT is better on only three instances, which are all very small wind farms from set $\mathcal{N}_1$, with a maximum ratio of 1.005. On sets $\mathcal{N}_2$, $\mathcal{N}_3$ and $\mathcal{N}_4$, the average ratio is within 0.84-0.86. On complete graphs, DIJNOR finds much better solutions than DIJPOT on all instances; the best ratio on $\mathcal{N}_5$ is 0.41, on 25.5 % of all instances DIJNOR computes costs that are only half as much as those computed by DIJPOT, on 61.5 % of all instances the ratio is smaller than 0.55.
Overall, DIJNOR outperforms DIJPOT on all benchmark sets, with an average ratio of 0.795. Hence, we choose DIJPOT as the better Dijkstra variant.

We briefly analyze why not using vertex potentials and reduced costs seems to be the far better strategy. As we have stated in the previous chapter when developing the SSP, the quality of the solution depends on the paths computed by the shortest path algorithm. We have also seen that Dijkstra does not necessarily find shortest paths in the presence of negative edges. This brings up the assumption that the quality of the computed paths may depend on the number of negative edges occuring. To examine this, we take a look at the ratio of negative edges occuring during the overall computation of a solution for

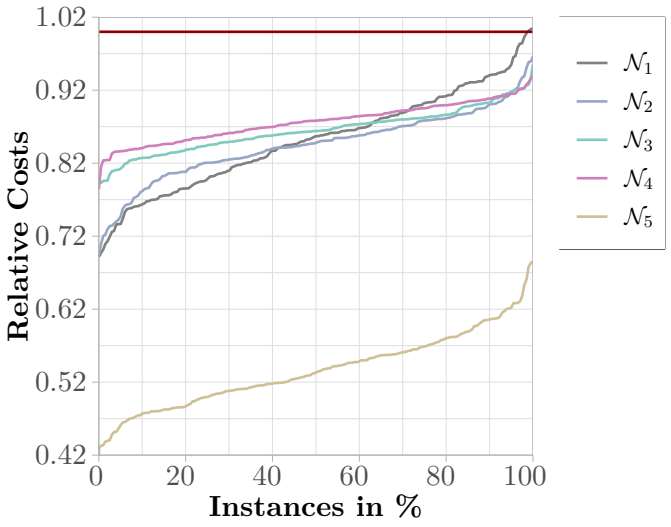

Figure 4.1: Relative cost ratios of DIJNOR compared to DIJPOT.

Table 4.2: Minimum, average and maximum negative edge ratios for each benchmark set in percent.

Strategy Combination	$\mathcal{N}_1$			$\mathcal{N}_2$			$\mathcal{N}_3$			$\mathcal{N}_4$			$\mathcal{N}_5$		
	min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
DIJNOR	2.37	3.19	4.49	2.50	3.04	3.74	2.59	2.89	3.29	2.47	2.76	3.02	0.12	0.22	0.35
DIJPOT	4.64	7.88	15.56	5.24	7.15	11.31	5.46	7.68	10.06	5.71	10.0	13.84	0.37	0.89	1.72

an instance, i.e. we divide the number of negative edges by the total number of edges throughout the computation. In Table 4.2 these ratios are displayed for both variants and for each benchmark set. We observe that DIJNOR has a significant better ratio on all benchmark sets, meaning much less negative edges occur compared to DIJPOT. The highest difference is found on $\mathcal{N}_5$: in average, using vertex potentials leads to over four times as many negative edges as using normal costs does. These observations seem to correlate with the quality of the found solutions by both variants and therefore with our assumption that more negative edges lead to worse solutions.
Now we give a possible explanation why vertex potentials lead to more negative edges. When not using vertex potentials, only edges with negative flow (i.e. their reverse edge has a positive flow) can have negative costs, since augmenting one unit of flow along them means to reduce the flow on its reverse edge, so eventually a cheaper cable type is sufficient for the resulting flow. In contrast, when using vertex potentials and reduced costs, the potential of a vertex affects all surrounding edges; that means, also edges with zero or positive flow can become negative. Our experiments have shown that most time, the majority of negative edges are zero flow edges, followed by edges with positive flow. This might explain why using vertex potentials leads to significantly more negative edges.
Looking at Table 4.2 also raises the question why on complete graphs ($\mathcal{N}_5$) the ratio of negative edges is much smaller than on the other wind farms. The reason for this is most likely because most of the edges on complete graphs are never used to carry flow. Therefore, the costs of these unused edges never become negative, especially when using normal costs.

4.1.2 Comparing BELNOR to BELPOT

We begin by comparing the running times of both strategies. Table 4.3 displays the minimum, average and maximum running times as well as the time ratios for each benchmark set. Both variants are fast, with running times ranging from tenth of milliseconds to just

Table 4.3: Minimum, average and maximum of running times in milliseconds of both compared variants. For each column, the minimum (i.e. the best) value is marked in green. The last row displays the minimum, average and maximum time ratios.

Strategy Combination	$\mathcal{N}_1$			$\mathcal{N}_2$			$\mathcal{N}_3$			$\mathcal{N}_4$			$\mathcal{N}_5$		
	min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
BELNOR	0.38	46.7	326	3.75	73.9	323	299	957	4.2k	5.6k	41.3k	156k	2.8k	17.0k	101.3k
BELPOT	0.38	46.7	328	3.74	74.1	310	311	966	4.3k	5.7k	43.0k	153k	2.8k	17.1k	103.4k
Time ratio	0.96	1.00	1.07	0.93	1.00	1.05	0.92	0.99	1.08	0.82	0.97	1.14	0.87	1.00	1.13

over two and a half minutes. On small wind farms (sets $\mathcal{N}_1$, $\mathcal{N}_2$ and $\mathcal{N}_3$), both strategies have very similar running times with average time ratios of 1.00, 1.00 and 0.99, and also the minimum and maximum running times differ only slightly. On large wind farms ($\mathcal{N}_4$), BELNOR is in average faster than BELPOT, though the time ratios spread between 0.82 and 1.14, meaning both strategies outrun the other on some instances. BELNOR is faster on 71.5 % of all instances on that set. On complete graphs, both variants have quite similar running times again, with BELNOR being slightly better.
Overall, BELNOR is faster on 61.1 % of all instances with an average time ratio of 0.990, indicating that there is no significant advantage over BELPOT. Therefore, we base our decision which strategy to choose on the quality of the solutions.

To compare the solution quality of both variants, we evaluate the relative cost ratio that is plotted in Figure 4.2. Our first observation is that both variants seem to find solutions of quite similar quality. On the small wind farms in benchmark set $\mathcal{N}_1$, they compute the same solution on all instances. On set $\mathcal{N}_2$, still on 61.5 % of all instances both strategies compute the same solution, on 24 % BELNOR finds better solutions. The ratios are widely spread, ranging from 0.919 to 1.056. On sets $\mathcal{N}_3$ and $\mathcal{N}_4$, the ratios do not spread that much and each strategy finds better solutions than the other on around $45 - 55$ %. The average ratio on both sets is 0.99, indicating that both strategies provide quite the same quality of solutions. On set $\mathcal{N}_5$, the maximum and minimum ratio spread wider again, with an average of 0.99. On 49 % of the instances, BELNOR finds better solutions than BELPOT, on 15 % both computed solutions are equal.
Overall, BELNOR finds better solutions on 35.6 % of all instances, in contrast BELPOT finds better solutions on only 27.5 %. Furthermore, the average ratio on each benchmark set, as well as the overall ratio, is less than one, meaning that BELNOR is slightly better than BELPOT. Hence, the strategy to use normal costs instead of vertex potentials and reduced costs is the better one in combination with Bellman-Ford.

4.1.3 Comparing DIJNOR to BELNOR

In the previous sections we found that for both Dijkstra and Bellman-Ford, using normal costs without vertex potentials is the better choice. In this section, we therefore compare DIJNOR and BELNOR to find the overall best SSP variant.
In Table 4.4 the running times are displayed as well as the time ratios. We already stated that DIJNOR always terminates in under 5 seconds while BELNOR requires up to just over two and a half minutes. In comparison, DIJNOR is faster than BELNOR on all instances. The larger the wind farms are, the more significant is the difference between both running times: while the average ratio is 0.16 on set $\mathcal{N}_1$, it is 0.02 on $\mathcal{N}_4$. On the latter, the minimum ratio is 0.01, meaning that DIJNOR requires only a hundredth of the time that BELNOR needs to terminate. Even on the smallest wind farms on set $\mathcal{N}_1$, DIJNOR is at least twice as fast as BELNOR.

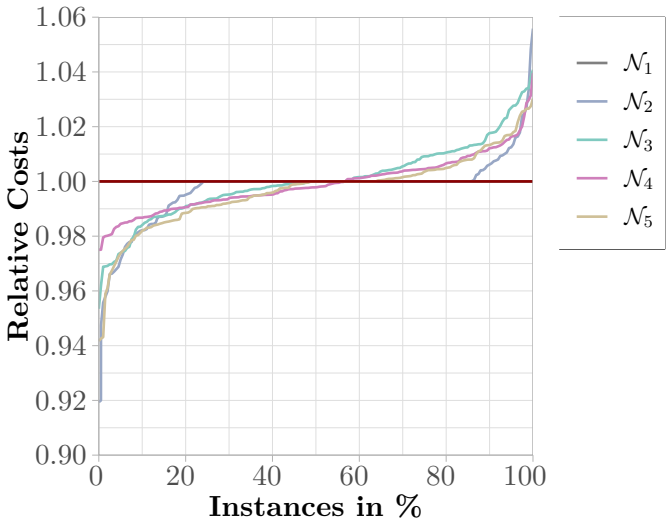

Figure 4.2: Relative cost ratios of BELNOR compared to BELPOT.

Table 4.4: Minimum, average and maximum of running times in milliseconds of both compared variants. For each column, the minimum (i.e. the best) value is marked in green. The last row displays the minimum, average and maximum time ratios.

Strategy Combination	$\mathcal{N}_1$			$\mathcal{N}_2$			$\mathcal{N}_3$			$\mathcal{N}_4$			$\mathcal{N}_5$		
	min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
DIJNOR	0.13	2.47	12.1	0.58	3.98	12.8	11.9	28.3	88.6	122	573	1.4k	167	872	4.8k
BELNOR	0.38	46.7	326	3.75	73.9	323	299	957	4.2k	5.6k	41.3k	156k	2.8k	17.0k	101.3k
Time ratio	0.04	0.16	0.36	0.04	0.08	0.18	0.02	0.03	0.04	0.01	0.02	0.02	0.04	0.05	0.06

Summarized, DIJNOR is much faster than BELNOR, especially on large wind farms. Still, also BELNOR terminates in a reasonable time.

To evaluate the quality of the solution, we look again at the relative cost ratio, plotted in Figure 4.3. On small wind farms ($\mathcal{N}_1$), both variants find equal solutions on 6.5 % of all instances, on 79 % DIJNOR computes better results. The ratios spread widely, ranging from 0.84 to 1.05. Similar results are observed on set $\mathcal{N}_2$, though there DIJNOR is better on 86.5 % of the instances and only on one instance both strategies find equal solutions. DIJNOR is even better on sets $\mathcal{N}_3$, $\mathcal{N}_4$ and $\mathcal{N}_5$ as it outperforms BELNOR on 98 % of the wind farms in these sets, with ratios ranging from 0.90 to 1.02. The average ratios do not differ significantly between the benchmark sets as they range from 0.962 to 0.970.
Overall, DIJNOR computes better solutions on 91.8 % of all instances with an average ratio of 0.968.

Altogether, we figured out that DIJNOR yields better solutions than BELNOR on each benchmark set and requires significant less time to compute those. Hence, we declare DIJNOR to be the best SSP variant.

4.2 Comparing our Best SSP Variant to NCC

Now we compare our best developed SSP variant DIJNOR to the NCC algorithm. For both algorithms, we evaluate the solutions provided after termination and the running times they need to terminate. From each benchmark set, we run the 200 instances that have been randomly selected in [GUW+19].

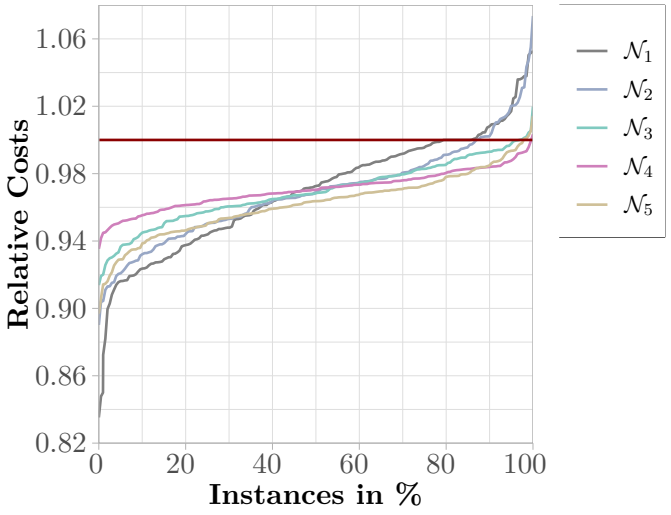

Figure 4.3: Relative cost ratios of DIJNOR compared to BELNOR.

Table 4.5: Minimum, average and maximum of running times in milliseconds of both compared variants. For each column, the minimum (i.e. the best) value is marked in green. The last row displays the minimum, average and maximum time ratios.

Strategy Combination	$\mathcal{N}_1$			$\mathcal{N}_2$			$\mathcal{N}_3$			$\mathcal{N}_4$			$\mathcal{N}_5$		
	min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
DIJNOR	0.14	2.62	12.2	0.57	3.85	12.2	12.3	28.7	95.0	125	548	1.4k	167	823	4.8k
NCC	0.47	33.7	279	2.99	51.7	260	159	666	3.2k	3.4k	26.1k	89.3k	1.8k	13.0k	97.4k
Time ratio	0.04	0.15	0.36	0.05	0.11	0.24	0.03	0.05	0.08	0.02	0.02	0.04	0.05	0.07	0.11

In Table 4.5 the minimum, average and maximum running times and time ratios are displayed. Our first observation is that our SSP is much faster than NCC on each wind farm instance. The running times of NCC range from half a millisecond to almost 100 seconds, while SSP always terminates in under five seconds. SSP outruns NCC especially on larger wind farms: while the average ratio on set $\mathcal{N}_1$ is 0.15, it is 0.02 on set $\mathcal{N}_4$, meaning that in average SSP is 50 times faster than NCC. The overall time ratio is 0.08.
Summarized, SSP is much faster than NCC, though also NCC provides reasonable running times since it terminates in under two minutes.

As in the previous sections, we evaluate the relative cost ratio of the two algorithms to compare their solution qualities; the ratios are plotted in Figure 4.4. As we see, NCC outperforms our SSP on each benchmark set. On $\mathcal{N}_1$, $\mathcal{N}_2$, $\mathcal{N}_3$ and $\mathcal{N}_4$ it computes better solutions on all instances. The ratios have a wide range on smaller wind farms: on $\mathcal{N}_1$ the minimum ratio is 1.014 and the maximum is 1.275; set $\mathcal{N}_2$ shows very similar results. On larger wind farms, the ratios spread less widely, ranging from 1.06 to 1.16 on both $\mathcal{N}_3$ and $\mathcal{N}_4$. The difference between NCC and SSP is more significant on smaller wind farms: while the average ratio on $\mathcal{N}_1$ is 1.154, it is only 1.101 on $\mathcal{N}_4$. On the latter set, our algorithm is within 10 % of NCC's solution on 53 % of the instances. On complete wind farms ($\mathcal{N}_5$), SSP computes better solutions than NCC on three instances with a minimum ratio of 0.973; though, the ratios have a wide range again, with a maximum of 1.244. The overall average ratio is 1.124.

In summary, the evaluation shows that NCC outperforms our algorithm on almost all instances in terms of solution quality, while our algorithm is much faster. Therefore,

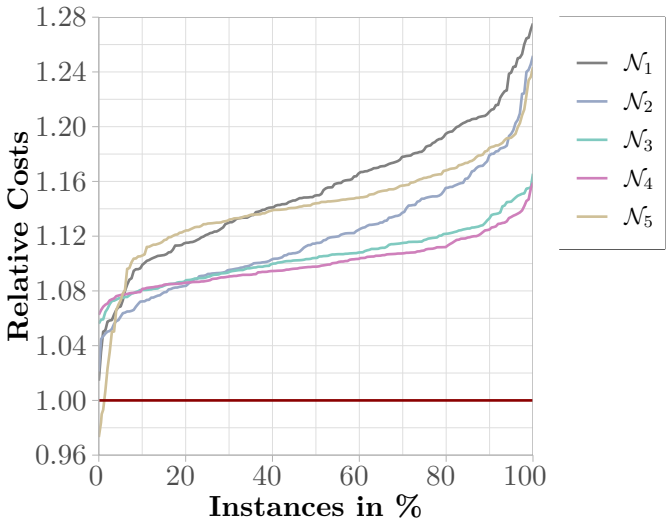

Figure 4.4: Relative cost ratios of DIJNOR compared to NCC.

NCC seems to be the better option in most cases. However, our algorithm can be a viable alternative in cases where very short running times are essential, for example in an interactive planning process. Especially on large wind farms it provides solutions much faster than NCC in a reasonable range of relative cost ratios.

4.3 Comparing our Best SSP Variant to MILP

We now compare our best developed SSP variant DIJNOR to the exact MILP solver *Gurobi* 8.0.0. The exact MILP formulation is found in [GUW+19]. From each benchmark set, we run the same 200 instances we used in the previous section. Other than our SSP, Gurobi does not only find feasible flows when it terminates, but can yield solutions during runtime. Since the computations of optimal solutions take too long on most instances (up to many days), we restrict the solver to different maximum running times and evaluate the solutions provided after these times. The maximum running times are between five seconds (which is above the maximum time the SSP takes to terminate) and one hour. For the SSP, we always use the solution found at termination. In addition to the comparison of the relative cost ratios (plotted in Figure 4.5), we compare the so called *relative gaps*, a standard notion from Mixed-integer Linear Programming. While computing solutions for an instance, Gurobi also tries to prove lower bounds (lb) on the optimal objective value for that instance. Together with the best found solution, referred to as upper bound (ub), the relative gap is computed as $^{\text{ub-lb}}/_{\text{ub}}$. The relative gap is in the unit interval and can indicate how far away the found solution is from the optimal objective value; higher values indicate worse solutions than lower values. In our evaluation, for each instance and maximum running time we use the lower bound found by Gurobi after one hour, and as the upper bound we use the best solution found by the SSP at termination and the best solution found by Gurobi after the specified maximum running time, respectively. We denote these relative gaps (plotted in Figure 4.6) as *SSP gap* and *MILP gap*.

After maximum running times of five seconds, our first observation is that Gurobi seems to be better on small wind farms, while SSP yields better solutions on larger wind farms (Figure 4.5). On $\mathcal{N}_1$, Gurobi finds better solutions than SSP on all but one instance, namely the largest wind farm of the benchmark set. The ratios spread widely from 0.95 to 1.27 with an average of 1.13. The SSP gaps are on average 28 % with a maximum of 38 % compared to an average MILP gap of 18.5 %, ranging from 0 % to 33.6 %. On $\mathcal{N}_2$, the

MILP gaps are on average 26.2 % with an outlier at 41.5 %. Repeatedly, the SSP gaps are worse with an average of 32.6 %. The relative costs ratios show similar results as on $\mathcal{N}_1$, though SSP performs slightly better than on $\mathcal{N}_1$ with a minimum of 0.86 and an average of 1.10. On $\mathcal{N}_3$, SSP is better on 18 % of the instances, the average ratio is 1.04. Both gaps are quite similar, with the MILP gaps having a slightly better average of 32.1 %, but also a wider distribution with a maximum of 39.4 %. The SSP gaps have a spread of only seven percentage points with a maximum of 38.5 %. On the large wind farms of set $\mathcal{N}_4$, SSP clearly outperforms the MILP: it finds better solutions on 81 % of the instances there. On seven instances, the ratio is less than 0.18, meaning that the computed flows cost only a fifth of those found by Gurobi. The average ratio on that set is 0.93. The SSP gaps are within a small range with a mean of 35.8 %. The MILP gaps are worse with an average of 40.5 % and seven instances above 88 %. On complete graphs ($\mathcal{N}_5$), SSP is better than Gurobi on all instances. The ratios range from 0.09 to 0.61 with an average of 0.15. On all but eight instances, SSP yields solutions that are at least 80 % cheaper than Gurobi's; all those eight instances contain only 80-90 turbines and therefore belong to the smallest wind farms of the set. The SSP gaps are on average 44.4 % and have a spread of ten percentage points, compared to an average of 91.9 % for the MILP gap. All but the above-mentioned eight instances have a MILP gap greater than 89 %. Altogether, we observe that within five seconds, Gurobi finds better solutions than SSP on small instances, but does not find reasonable results on large and especially on complete graphs. On the latter, SSP provides better results.

After maximum running times of five minutes, we observe different results: Gurobi finds better solutions on all instances. While the minimum ratios are quite similar across all benchmark sets, ranging from 1.02-1.05, the maximum and average ratios are greater on smaller wind farms. On $\mathcal{N}_1$, the maximum ratio is 1.27, on $\mathcal{N}_4$ it is only 1.15. The average ratios range between 1.16 ($\mathcal{N}_1$) and 1.08 ($\mathcal{N}_4$). Especially on complete graphs, Gurobi is now able to find way better results than it did after five seconds and outperforms the SSP with an average ratio of 1.13. The MILP gaps are much more consistent after five minutes compared to those after five seconds. On $\mathcal{N}_2$, the maximum decreases to 29.6 %, the outlier is not present anymore. On $\mathcal{N}_4$, the MILP gaps now have a mean of only 29.7 % and a maximum of 32.3 %. The most significant change is found on $\mathcal{N}_5$: the average gap is now 36.3 %, compared to the mean of 91.9 % after five seconds; the maximum gap after five minutes is 42 %. The SSP gaps (that did not change since the solution provided after 5 seconds is the final solution) are worse than the MILP gaps on each instance now.

With even longer maximum running times of up to one hour, the results look vastly the same: Gurobi improves its found solution only slightly, the most noteworthy difference takes place on the larger wind farms. After fifteen minutes, the average relative cost ratio of both $\mathcal{N}_4$ and $\mathcal{N}_5$ increases by one percentage point. After one hour, the average ratio of $\mathcal{N}_5$ increases again to 1.15 (compared to 1.13 after five minutes). On the other sets, the only notable changes are small increases of the minimum and the maximum ratio. The average ratio of all instances after five minutes is 1.1179, after fifteen minutes 1.1224 and after one hour 1.1257, underlining that there are only slight differences between the maximum running times. According to that, also the MILP gaps after one hour look extensively the same as after five minutes.

Summarized, the experiments have shown that the SSP is a viable option on large wind farms when maximum running times of few seconds are essential. In such short times, Gurobi does not find reasonable solutions on most instances, while our algorithm does. With more time available, the SSP cannot compete with Gurobi as the latter profits from longer running times and yields better solutions on all instances.

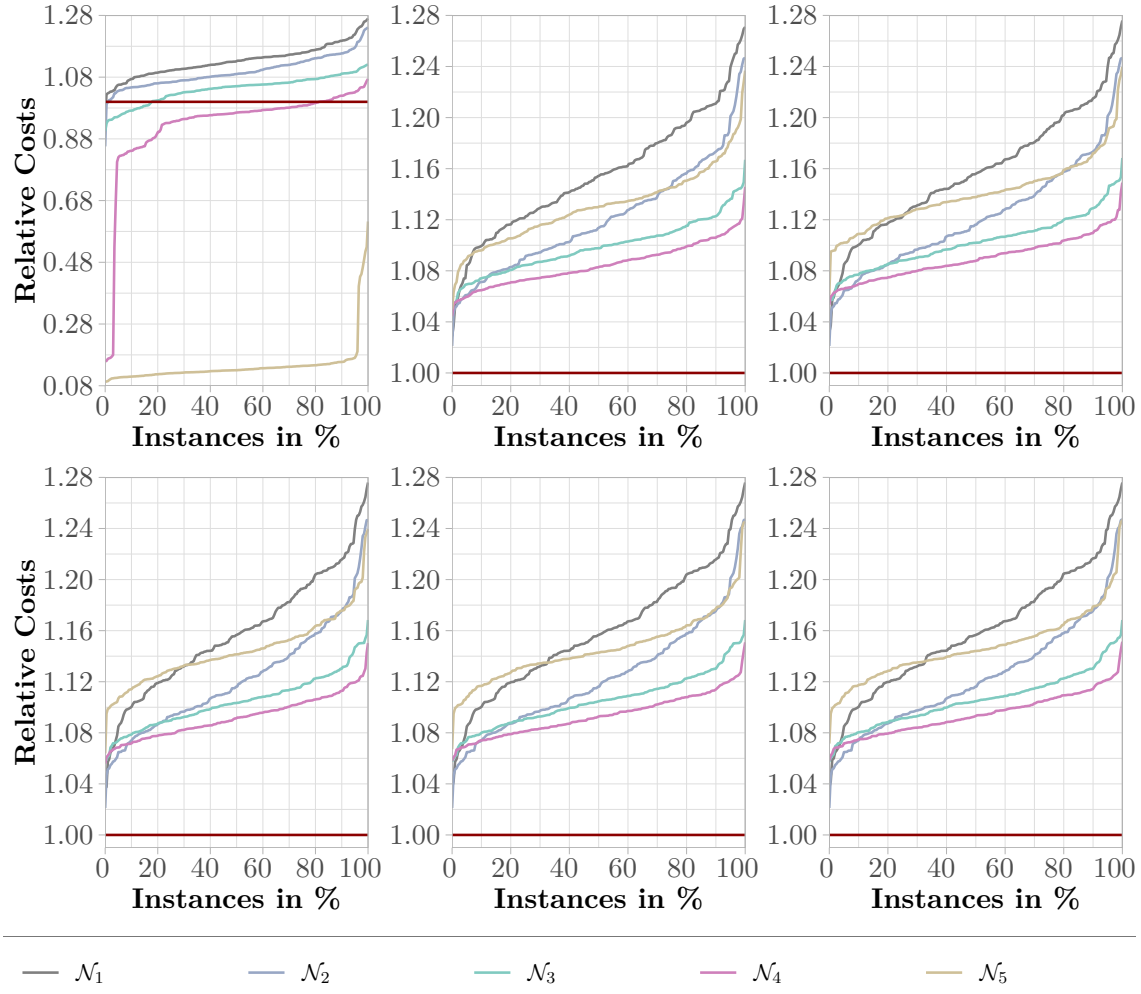

Figure 4.5: Relative cost ratios of the SSP compared to Gurobi at different maximum running times. Upper left to upper right: 5 s, 5 min, 15 min; bottom left to bottom right: 30 min, 45 min, 1 h.

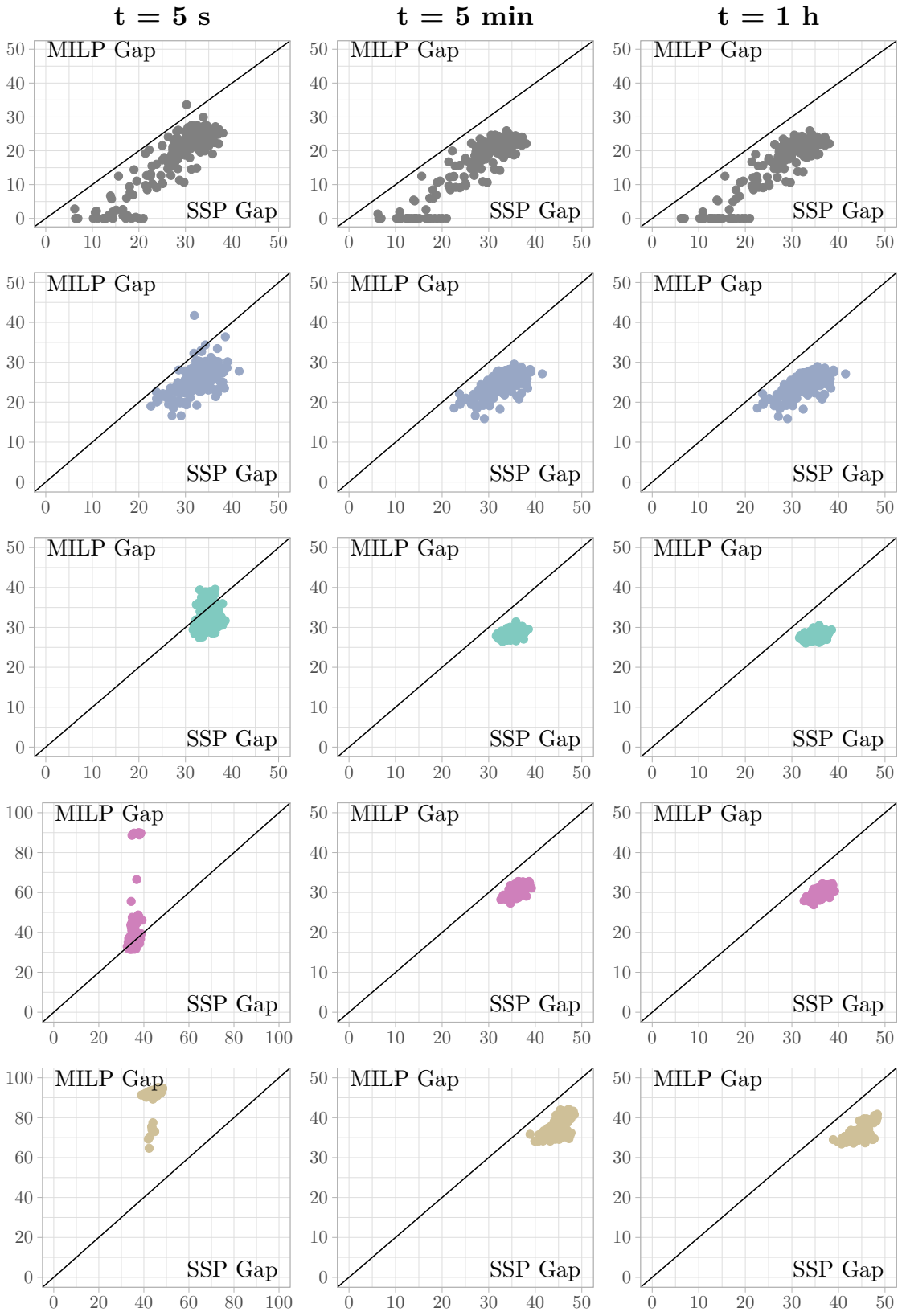

Figure 4.6: Comparison of SSP and MIP gaps for each benchmark set (rows 1-5 show $\mathcal{N}_1$-$\mathcal{N}_5$) after maximum running times of five seconds, five minutes and one hour.

5. Conclusion

In this thesis, we examined algorithms for the Wind Farm Cabling Problem. For this purpose, we modeled the WCP as a flow network and looked at its similarities and differences to the well-known Minimum-Cost Flow Problem. We then looked at various optimal MCF algorithms and analyzed whether we can adapt these algorithms to solve the WCP. For the Primal-Dual Algorithm, the Out-Of-Kilter Algorithm and the Relaxation Algorithm we found that serious problems arise due to the non-linear step cost function of the WCP. These problems led to our decision that none of these are appropriate for the WCP and therefore have not been developed further.

For the Successive Shortest Path Algorithm, we found that an adaption to a WCP algorithm is suitable. We developed two possibilities concerning the usage of vertex potentials as well as two strategies of computing short paths, resulting in four variants of the algorithm. In the next step, we compared these variants among each other to find the best one in terms of running times and quality of the found solutions.
Thereafter we compared our best variant of the SSP to already existing WCP algorithms, namely a negative cycling canceling heuristic and a MILP solver. We found that our algorithm provides reasonable solutions in very short running times, but cannot keep up with the other algorithms in terms of solution quality after longer running times. Therefore, our algorithm can be a viable option in time-critical planning processes especially on large wind farms, since it computes good solutions within a few seconds whereas NCC and MILP need significantly more time to find comparable solutions.

5.1 Further Work

Regarding our developed SSP algorithm, one may examine ways to improve the quality of the found solutions. The probably most promising approach for this may be improving the computed short paths. Both our heuristics, Dijkstra and Bellman-Ford, do not find the shortest simple paths in the presence of negative edges. Thus, one may develop heuristics that compute better paths than our heuristics do; this will most likely come along with a trade-off between running time and quality. One can also go deeper into our claim that more negative edges lead to worse solutions; if this can be proved to be true, another way of improving the quality of computed short paths can be to reduce the number of negative edges occurring throughout the SSP.
Another approach is to identify and evaluate different strategies of choosing the start vertex

in each iteration. Until now, we simply select the turbines in the order they appear in the list representation of the wind farm. Another possible strategy may be to base the selection on the geographic distances between turbines and their nearest substations.

Another approach worth trying may be combining our algorithm with NCC or MILP. Since the latter two need to start with a feasible initial flow, and our algorithm provides such a flow in only a few seconds, the SSP's solution could be used as warm start for NCC or MILP. This may result in faster computations and better solutions compared to the currently used initial flow.

Bibliography

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[Bel58] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[BG60] Robert G. Busacker and Paul J. Gowen. A procedure for determining a family of minimum-cost network flow patterns. 1960.

[BVMO16] Constantin Berzan, Kalyan Veeramachaneni, James McDermott, and Una-May O'Reilly. Algorithms for Cable Network Design on Large-scale Wind Farms. `http://thirld.com/files/msrp_techreport.pdf`, 2016. Accessed: 2019-12-02.

[Dij59] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM*, 19(2):248–264, April 1972.

[Ful61] Delbert Ray Fulkerson. An out-of-kilter algorithm for minimal cost flow problems. *Journal of The Society for Industrial and Applied Mathematics*, 9:18–27, 1961.

[GKK74] F. Glover, D. Karney, and D. Klingman. Implementation and computational comparisons of primal, dual and primal-dual computer codes for minimum cost network flow problems. *Networks*, 4(3):191–212, 1974.

[GUW+18] Sascha Gritzbach, Torsten Ueckerdt, Dorothea Wagner, Franziska Wegner, and Matthias Wolf. Towards negative cycle canceling in wind farm cable layout optimization. In *Proceedings of the 7th DACH+ Conference on Energy Informatics*, volume 1 (Suppl 1). Springer, 2018.

[GUW+19] Sascha Gritzbach, Torsten Ueckerdt, Dorothea Wagner, Franziska Wegner, and Matthias Wolf. Engineering Negative Cycle Canceling for Wind Farm Cabling, 2019.

[Iri60] Masao Iri. A new method for solving transportation-network problems. 1960.

[Jew62] William S. Jewell. Optimal flow through networks. 1962.

[JM93] David S. Johnson and Catherine C. McGeoch. Network flows and matching: First dimacs implementation challenge. 1993.

[KK12] Z. Király and P. Kovács. Efficient implementations of minimum-cost flow algorithms, 2012.

[Kov15] Péter Kovács. Minimum-cost flow algorithms: An experimental evaluation. *Optimization Methods and Software*, 30, 01 2015.

[LR13] S. Lumbreras and A. Ramos. Optimal design of the electrical layout of an offshore wind farm applying decomposition strategies. *IEEE Transactions on Power Systems*, 28(2):1434–1441, 2013.

[LRWW17] Sebastian Lehmann, Ignaz Rutter, Dorothea Wagner, and Franziska Wegner. A simulated-annealing-based approach for wind farm cabling. In *Proceedings of the Eighth International Conference on Future Energy Systems*, e-Energy '17, pages 203–215, New York, NY, USA, 2017. ACM.

[Min60] G. J. Minty. Monotone networks. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 257, 09 1960.

[Orl97] James B. Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming*, 78:109–129, 1997.

[Sch03] Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, 2003.

[Sta18] Dominik Stampa. Verkabelung von Windfarmen auf Bäumen. Bachelor thesis, Karlsruhe Institute of Technology, September 2018.

[Tar85] Éva Tardos. A Strongly Polynomial Minimum Cost Circulation Algorithm. *Combinatorica*, pages 247–255, 1985.

[win] Wind energy in Europe in 2019 trends and statistics. `https://windeurope.org/wp-content/uploads/files/about-wind/statistics/WindEurope-Annual-Statistics-2019.pdf`. Accessed: 2020-24-04.