Karlsruhe Institute of Technology

Clusteringansätze von Windfarmen

Bachelorarbeit
von

Hannah Wenk

An der Fakultät für Informatik
Institut für Theoretische Informatik

Erstgutachter:	Prof. Dr. Dorothea Wagner
Zweitgutachter:	Prof. Dr. Peter Sanders
Betreuende Mitarbeiter:	Franziska Wegner
	Michael Hamann

Bearbeitungszeit: 23. Februar 2017 – 22. Juni 2017

Statement of Authorship

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

Karlsruhe, 22. Juni 2017

Abstract

In my thesis I implemented algorithms to compute the cable layout problem in wind farms with graph clustering. This is the problem of how to connect the different components of a wind farm (the turbines and substations in particular) with one another. The problem is complicated by the existence of different cables which can accomodate a different number of turbines. I implemented several variants of a mover algorithm for this and compared them to the results from a Multilevel Quality Threshold Clustering and Mixed Integer Linear Program. These mover algorithms got overall similar results than the MILP but were much faster, even instances with 300 turbines are usually solved in under 15 minutes.

Deutsche Zusammenfassung

In meiner Bachelorarbeit habe ich Algorithmen zur Berechnung Kabellayoutproblems für Windfarmen mit Graphclustering entwickelt. Dies bedeutet die Verkabelung innerhalb einer Windfarm zwischen den Windenergieanlagen (WEA) und dem Umspannstationen, die Problematik besteht dabei sowohl darin welche Komponenten miteinander verbunden werden als auch welche Kbael verwendet werden. Dabei habe ich mehrere Varianten eines Moveralgorithmus implementiert und mit den Ergebnissen durch Multilevel Quality Threshold Clustering sowie durch ein Mixed Integer Linear Program verglichen. Die Moveralgorithmen waren dabei ähnlich gut wie das MILP dabei aber deutlich schneller, Instanzen bis 300 WEA konnten in unter 15 Minuten gelöst werden.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Verwandte Arbeiten . . . 2
1.2 Überblick . . . 3

2 Grundlagen **5**
2.1 Einführung in Graphen . . . 5
2.2 Clustering . . . 7
2.2.1 Graph-Clustering . . . 9

3 Windfarmodellierung **11**

4 Untersuchte Clusteringalgorithmen **15**
4.1 Moveralgorithmus . . . 15
4.1.1 Standard-Mover . . . 16
4.1.2 Single-Mover . . . 17
4.1.3 Adoptionsmover . . . 18
4.1.4 Multilevel Quality Threshold . . . 19
4.1.4.1 Multilevel Quality Threshold mit Kostenfunktion . . . 20
4.2 Fallstudien . . . 20
4.2.1 Daten . . . 20
4.2.2 Durchführung . . . 21

5 Zusammenfassung **33**
5.1 Zukünftige Arbeit . . . 34

Literaturverzeichnis **35**

1. Einleitung

Erneuerbare Energien werden immer wichtiger für die Stromerzeugung aus verschiedenen Gründen. Bisher wird immer noch mehr als die Hälfte des Stroms in Deutschland durch fossile Energien produziert [str17]. Fossile Energien haben aber mehrere Nachteile, zum einen sind sie endlich und werden damit langfristig teuer werden, sowie Grund für Konflikte bieten. Zum anderen produziert die Verbrennung von fossilen Energieträgern Kohlendioxid, dies ist das hauptverantwortliche Gas für den Treibhauseffekt und den Klimawandel. Um den Klimawandel eindämmen zu können, muss deshalb die Verbrennung von fossilen Energieträgern stark reduziert werden. Deshalb sollen Erneuerbare Energien gefördert werden, weltweit wird dies durch das Klimaabkommen von Paris [Nat15] gefördert. In Deutschland ist die Förderung erneuerbarer Energien schon seit 2000 durch das Gesetz für den Ausbau erneuerbarer Energien sehr erfolgreich. Eine der wichtigsten Erneuerbaren Energien ist Windkraft. Für effektive Windnutzung eignen sich besonders große Windfarmen, sowohl auf dem Land (On-Shore) als auch im Meer (Off-Shore) Windfarmen werden immer größer, ein Beispiel hierfür ist der Windpark Gansu, der in der aktuellen Ausbaustufe schon bis zu 6 000MW produzieren kann und aus 3 500 Windenergieanlagen in 18 unterschiedlichen Windfarmen besteht. Bei solchen großen Windfarmen ist das Layout der Kabel zwischen den einzelnen Komponenten, immer wichtiger und gleichzeitig immer komplexer.
Wie die Zeichnung in 1.1, zeigt besteht eine Windfarm mindestens aus den eigentlichen Windenergieanlagen (WEA), umgangsprachlich auch Turbinen oder Windrad genannt, den Umspannstationen, an die die WEA angeschlossen sind, einem oder mehreren Umspannwerken, für den Anschluss der Umspannstationen an das Stromnetz, sowie den Kabeln zwischen all diesen Komponenten.
Die Verkabelungsinfrastruktur von Windfarmen ist unterschiedlich bei kleineren und größeren Windfarmen. Für kleinere Windfarmen werden die WEA meist mit Niederspannungsanlagen miteinander verbunden, die Farm wird dann über einen Transformator und einer Mittelspannungsschaltanlage sowie der Mittelspannungsanschlussleitung bis zum Netzanbindungspunkt angeschlossen. Bei größeren Windparks mit WEA in der Multimegawatt-Klasse, die jeweils mit einem eigenen Transformator ausgestattet sind, ist die interne Verkabelung auf Mittelspannungebene und wird an einem Umspannnwerk an das (Hochspannungs)-Stromnetz angeschlossen.

In meiner Arbeit möchte ich das Problem des Kabellayouts in einer Windfarm untersuchen, damit meine ich die Verkabelung der WEA und Umspannstationen miteinander. Genauer gesagt untersuche ich die Kosten eine solche Verkabelung zu installieren. Die Platzierung der WEA ist abhängig von den Windverhältnissen am Ort, dem Abstand der WEA

voneinander und der Geographie. In meiner Arbeit gehe ich allerdings davon aus, dass die Orte und Typen der WEA und Umspannstationen schon vorher festgelegt wurden. Da die Umspannstationen fix sind, können die Kabel zwischen Umspannstationen und Umspannwerken vernachlässigt werden. Die Kosten können also reduziert werden auf die Grabungsarbeiten sowie die Kabelkosten der Kabel zwischen WEAs sowie zwischen WEA und Umspannstationen. Dabei kosten Kabel, die auf einen höheren Nennstrom ausgelegt sind, mehr pro Längeneinheit als solche für einen niedrigeren Nennstrom. Die Frage ist also welche WEA miteinander, bzw. mit der Umspannstation verbunden werden und welche Kabel verwendet werden. Dabei müssen sowohl die Kapazitäten der Kabel, bedingt durch den Nennstrom dieser Kabel, als auch die Kapazitäten der Umspannstationen, bedingt durch die Nennleistung der Umspannstationen, eingehalten werden.

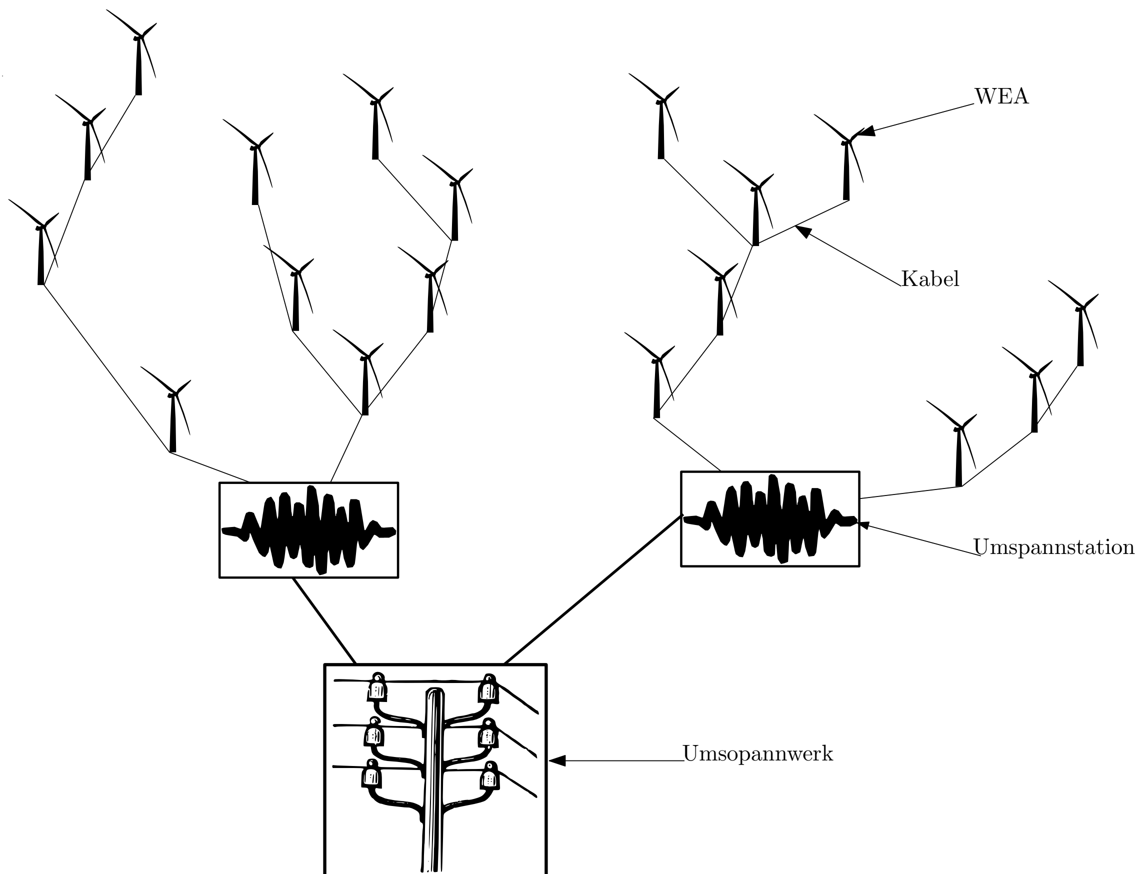

Abbildung 1.1: Vereinfachte Zeichnung einer Windfarm

1.1 Verwandte Arbeiten

Bisher wurde die Planung von Windfarmen meist ohne algorithmische Ansätze und stattdessen mit Erfahrungswerten gelöst. Wenn man nur einen Kabeltypen verwendet, ließe sich dieses Problem mit Heuristiken für beschränkte minimale Spannbäume (CMST von Capacitated Minimum Spanning Tree) lösen. Es existieren einige Ansätze dieses Problem auch bei mehreren Kabeltypen algorithmisch zu lösen.
So verwenden Dutta und Overbye [DO11] einen Ansatz mit mehreren Ebenen an Quality Threshold Clustering. Dabei werden allerdings nur Windfarmen mit einer Umspannstation betrachtet.
Die Idee hierbei ist mehrfach Quality Threshold Clustering mit unterschiedlichen WEA und wachsendem Threshold auszuführen. Die Idee bei Quality Threshold Clustering ist es Datenpunkte zusammenzufassen, die näher als ein gegebener Schwellwert zueinander

sind. Im Detail wird Quality Threshold Clustering im Abschnitt 2.2 erklärt. Bei Multilevel Quality Threshold werden zunächst alle WEA mit einem relativ kleinen Schwellwert geclustert. Pro gefundenem Cluster wird die WEA, deren Distanz zur Umspannstation am kleinsten ist, als „First Level Cluster Representative Point"ausgezeichnet. Alle anderen WEA aus diesem Cluster werden sternförmig mit dem First Level Cluster Representative Point verbunden.
Für das nächste Level werden alle Cluster Representative Points des vorherigen Levels mit dem doppelten des vorherigen Schwellwerts geclustert und wieder die WEA, die am nächsten an der Umspannstation ist, bestimmt und als „Second Level Cluster Representative Point"ausgezeichnet. Die anderen WEA in diesem Cluster werden wieder mit diesem Second Level Cluster Represenative Point sternförmig verbunden. Dies wird so lange wiederholt, bis die Anzahl an WEA im größten Cluster kleiner als ein bestimmtes Toleranzlevel ist, oder die Kabelkapazität erreicht ist.
Ein anderer Ansatz wurde von Berzan et al. [BVMO11] gewählt. Dabei wird das Kabellayout mit einem Minimalen Spannbaum gelöst. Außerdem wird auch vorgeschlagen, das Problem in 3 Unterprobleme aufzuteilen:

- **Circuit Problem:** Gegeben ist die Menge an WEA, die einen Stromkreis bilden, also durch Kabel miteinander verbunden sind und über exakt eine WEA an die Umspannstation angeschlossen sind. Das Problem besteht darin eine möglichst günstige Verkabelung innerhalb dieser Menge zu finden.
- **Substation Problem** Gegeben ist eine Umspannstation sowie alle an diese angeschlossene WEA. Das Problem besteht darin eine Aufteilung der WEA in Stromkreise zu finden und innerhalb der Stromkreise eine Verkabelung zwischen den WEA zu finden.
- **Full Farm Problem** DIes ist das volle Problem, also mit mehreren Umspannstationen und vielen WEA. Das Problem besteht dann darin eine Zuordnung der WEA zu Umspannstationen zu finden und innerhalb dessen eine Zuordnung dieser WEA zu Stromkreisen sowie schlussendlich eine möglichst günstige Verkabelung innerhalb dieser Stromkreise.

Diese Ansätze betrachten meistens nur relativ kleine Windfarmen und/oder eine kleine Auswahl an Benchmarkdatensätzen. Ein Ansatz um mit einer Vielfalt von unterschiedlich großen Windfarmen umzugehen gab Lehmann et. al [LRWW17]. Darin wird ein Simulated-Annealing-Algorithmus beschrieben, der mit der gleichen Aufteilung wie Berzan et al.[BVMO11] das Problem löst. Außerdem stehen damit zum ersten Mal eine große Anzahl an Benchmark Windfarmen in verschiedenen Größen zur Verfügung.

1.2 Überblick

Ein wichtiges Problem beim Design von großen Windfarmen ist das Kabellayout und damit zusammenhängend die Zuordnung von Windfarmen an Umspannstationen bzw. Schaltkreise. In dieser Arbeit möchte ich versuchen dieses Problem durch verschiedene Clusteringverfahren zu lösen. Dazu wurden mehrere Varianten eines „Mover"-Algorithmus verglichen. Dieser Mover-Algorithmus ist ein hierarchisches Graphclusteringverfahren, dass die Kosten der verschiedenen Kabel sowie die Kapazitäten berücksichtigt. Dieser Algorithmus wurde gewählt, da zum eine Modellierung einer Windfarm für dieses Problem als Graph naheliegend ist und damit graphbasierte Verfahren sich besser anbieten um die komplexere Kostenfunktion abzubilden als Clusteringverfahren, die die WEA als einzelne Datenpunkte betrachten, und zum anderen bei der komplexen Kostenfunktion ein iterativer Algorithmus sehr gut geeignet ist. Außerdem wurden der Algorithmus von Dutta et. al. [DO11] sowie eine Verbesserung davon zum Vergleich ebenfalls implementiert. Schlussendlich

wurde auch ein Kabellayout basierend auf einem Minimalen Spannbaum betrachtet. Alle Ergebnisse wurden dann verglichen mit den Ergebnissen durch ein Mixed Integer Linear Programm.

2. Grundlagen

Eine Windfarm besteht aus mehreren einzelnen Windenergieanlagen (WEA) und der dazugehörigen Infrastruktur. Diese Infrastruktur besteht aus den Umspannstationen an denen die Kabel der Windenergieanlagen gesammelt und die Spannung geändert wird, sowie weitere Regelungsaufgaben übernommen werden. Außerdem haben große Windfarmen auch meistens noch einen Netzübergabepunkt, der die Umspannstationen mit dem Stromnetz verbindet. Die Windenergieanlagen in einer Windfarm sind oft hierarchisch organisiert. Die kleinste Einheit über einer Windenergieanlage ist dabei ein **Stromkreis**, also miteinander verbundene Komponenten die über einen Punkt an die Sammelschiene der Umspannstation angeschlossen sind. Mehrere Stromkreise sind dann zu einem **Sammelsystem** an einer Umspannstation zusammengefasst. Die Umspannstationen sind evtl. über mehrere weitere Umspannstationen an einen Netzübergabepunkt angeschlossen.
Die üblichen Layouts einer Windfarm sind dabei nach Martin Kaltschmitt [Kal13] eine **radiale** Anordnung, bei der die Windenergieanlagen in eine Reihe hintereinander verbunden werden, eine **sternförmige** Verknüpfung, bei der die Windenergieanlagen mit einer zentral gelegenen Windenergieanlage verbunden sind, sowie die **Ringanordnung** bei der die Anlagen in offenem Ringbetrieb geschaltet werden, also in zwei Halbringen betrieben werden, die im Fehlerfall zusammen geschaltet werden können, so dass die restlichen Anlagen auch dann noch mit der Umspannstation verbunden sind. Eine Skizze dieser drei Varianten ist in 2 zu finden.

2.1 Einführung in Graphen

Diese Einführung basiert in großen Teilen auf dem Graphen Skript von Prof. Dr. Wagner [Wag98]. Ein **gerichteter, kantengewichteter Graph** ist ein Tripel $G = (V, E, \omega)$. Dabei ist V eine nichtleere Menge der Knoten des Graphen. Die Menge geordneter Knotenpaare von Kanten heißt $E \subseteq V \times V$, $\omega : E \to \mathbb{R}$ ist eine Funktion, die jeder Kante $e \in E$ ein Gewicht $\omega(e)$ zuweist.
Zwei Knoten $v, w \in V$ heißen **benachbart** oder **adjazent**, wenn sie durch eine Kante $(v, w) \in E$ verbunden sind. Die Menge $N(v)$ bezeichnet die Menge der Nachbarn des Knoten v. Bei einem gerichteten Graphen heißt der Knoten $x \in V$ **Vorgänger** von $y \in V$ falls (x, y) eine gerichtete Kante von G ist, entsprechend heißt der Knoten y Nachfolger von x. Eine Kante $e \in E$ heißt **inzident** zu einem Knoten v, falls dieser Knoten einer der Endpunkte der Kante ist. Die Menge aller zum Knoten v inzidenten Kanten bezeichne $\delta(v)$. Der **Knotengrad** $d_G(v)$ eines Knoten v ist die Anzahl aller Nachbarn des Knoten v. Der

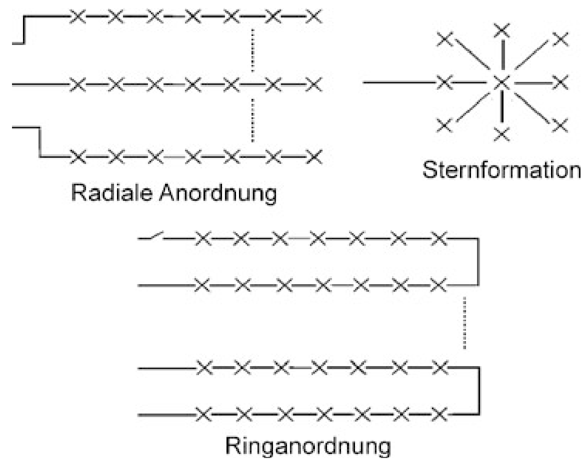

Abbildung 2.1: Die verschiedenen üblichen Kabellayouts nach Martin Kaltschmitt Erneuerbare Energien : Systemtechnik, Wirtschaftlichkeit, Umweltaspekte [Kal13]

Eingangsgrad $d_G^-(v)$ des Knotens v ist die Anzahl aller Vorgänger des Knotens, analog dazu ist der Ausgangsgrad $d_G^+(v)$ die Anzahl der Nachfolger des Knotens v. Ein **Teilgraph** $H = (V_H, E_H)$ eines Graphen $G = (V, E)$, falls eine injektive Abbildung $f : V_H \rightarrow V$ existiert, so dass gilt $\{u, v\} \in E_H \Rightarrow \{f(u), f(v)\} \in E$. Ein Teilgraph H heißt **knoteninduziert** durch V_H, falls er alle Kanten von G enthält, die beide Endpunkte in V_H haben, d.h. $E_H = \{\{u, v\} \in E : u, v \in V_H\}$. Ein Teilgraph heißt aufspannend, falls $V(H) = V(G)$. Ein **Weg** ist eine Folge $v_0, v_1, \ldots, v_l$ von (nicht notwendigerweise verschiedenen Knoten $v_i \in V, 0 \leq i \leq l$, wobei entweder $l = 0$ oder für $i = 0, \ldots, l-1$ gilt $\{v_i, v_{i+}\} \in E$. Die Länge eines Pfades ist die Anzahl l der Knoten, die er enthält. Ein Kreis ist ein Weg $W = v_0, v_1, \ldots, v_l$ bei dem $v_0 = v_l$.
Ein Graph G heißt **zusammenhängend**, wenn zwischen allen zwei Knoten $u, v \in V$ ein Weg existiert. Ein **Baum** ist ein kreisfreier, zusammenhängender Graph T und es gilt $|E| = |V| - 1$. Ist T kreisfrei, aber nicht unbedingt zusammenhängend, dann heißt T **Wald**. Ist ein Baum T Teilgraph eines Graphen G mit $V(T) = V(G)$, so heißt T **aufspannender Baum**. Ist T gerichtet und es gibt einen ausgezeichneten Knoten $r \in V$, der Wurzel genannt wird, dann wird der Baum **In-Tree** genannt wenn, alle Kanten eine Orientierung haben so dass für jeden Knoten $v \in V \neq r$ ein Pfad existiert, der von v zu r gerichtet ist, ein **Out-Tree** ist entsprechend ein Baum bei dem alle Pfade von r zu v gerichtet sind. Bei einem Out-Tree hat die Wurzel r also Eingangsgrad $d_G^-(r) = 0$, bei einem In-Tree entsprechend Ausgangsgrad $d_G^+(r) = 0$. Als **Höhe** eines Baumes T bezeichnet man die Länge des längsten Pfades innerhalb von T. Bei einem Out-Tree werden alle Knoten mit Ausgangsgrad 0 als **Blätter** bezeichnet. Alle Knoten, die nicht Blätter sind, heißen **Innere Knoten**. Für jeden Knoten $v \in V \neq r$ wird der Vorgänger von v als **Elter** oder **Elternknoten** bezeichnet, die Nachfolger eines Knoten $v \in V$ heißen umgekehrt **Kinder** von v.

Minimaler Spannbaum (MST)

Ein MST ist ein aufspannender Baum mit minimalem Gewicht. Die Definition des Problems ist die Folgende:

Definition 2.1. *Gegeben sei ein zusammenhängender Graph $G = (V, E)$ und eine Gewichtsfunktion $\omega : E \rightarrow \mathbb{R}$. Finde einen aufspannenden Baum $B = (V, E')$ von G mit $E' \subseteq E$, der bezüglich ω minimales Gewicht hat. Das heißt so dass: $\omega(B) = \sum_{\{u,v\} \in E'} \omega(\{u, v\})$*

minimal über alle aufspannenden Bäume in G ist.

Einer der Standardalgorithmen um dieses Problem zu lösen ist der Algorithmus von Prim [Kru12], dieser Algorithmus lässt den MST von einem Knoten $v \in V$ wachsen. Der Algorithmus startet mit der MST-Kantenmenge $E_T := \emptyset$ und der MST-Knotenmenge $T := \{v\}$. Solange T noch nicht alle Knoten aus V enthält, wähle eine Kante $e \in E$ mit minimalem Gewicht aus, die inzident zu einen Knoten $u \in V$ mit $U \notin T$. Füge diese Kante e und den Knoten u zu T hinzu. Die Laufzeit dieses Algorithmus liegt bei $\mathcal{O}(|E| + |V| \log |V|)$.

Flüsse und Flussnetzwerke

Ein weiteres wichtiges Konzept in meiner Arbeit ist der Fluss auf einem Graphen.

Definition 2.2. *Gegeben sei ein einfacher gerichteter Graph $D = (V, E)$ mit Kantenkapazitäten $c : E \to \mathbb{R}_0^+$ und ausgezeichneten Knoten $s, t \in V$, wobei s die **Quelle** (source und t die **Senke** (target) ist. Das Tupel $N = (D; s; t; c)$ wird dann **Netzwerk** genannt. Eine Abbildung $f : E \to \mathbb{R}_0^+$ heißt **Fluss**, wenn sie die folgenden Bedingungen erfüllt:*

- *Für alle $\{j, i\} \in E$ ist die folgende **Kapazitätsbedingung** erfüllt.*

$$0 \leq f(i, j) \leq c(i, j)$$

- *Für alle $i \in V \backslash \{s, t\}$ ist die folgende **Flusserhaltungsbedingung** erfüllt:*

$$\sum_{\{j|(i,j) \in E\}} f(i, j) - \sum_{\{j|(i,j) \in E\}} f(j, i) = 0$$

2.2 Clustering

Das Clustering von (großen) Datenbeständen ist die Suche nach Ähnlichkeitsstrukturen in diesen Datenbeständen. Eine Gruppe von „ähnlichen“Objekten wird dabei **Cluster** genannt. Wenn Datenpunkte geclustert werden, kann als Ähnlichkeitsmaß zum Beispiel der Abstand zwischen den Objekten verwendet werden. Gebräuchliche Algorithmen für diese Art des Clusterings sind zum Beispiel **k-Means** oder **Quality Threshold Clustering**.

k-Means

Dieser Algorithmus wurde zuerst von Steinhaus [Ste57] vorgestellt. Die folgende Beschreibung basiert auf [Mac11].

Die allgemeine Idee des Algorithmus ist es, den Datensatz so in k Partitionen $S = \{S_1, S_2, \ldots, S_k\}$ aufzuteilen, dass die Summe der quadrierten Abweichungen vom Clusterschwerpunkt, gemessen in der euklidischen Distanz minimal ist. Es wird also der Schwerpunkt (engl. means) der Datenpunkte je Cluster berechnet. Dies bedeutet also:

$$J = \sum_{i=1}^{k} \sum_{x_j \in S_i} \|x_j - \mu_i\|^2$$

Wobei x_j die Datenpunkte und μ_i die Schwerpunkte der Cluster S_i sind. Die Anzahl k an Clustern muss dabei schon von Beginn an feststehen.
Zu Beginn werden die k Schwerpunkte $\mu_1, \ldots \mu_k$ zufällig gewählt. Der eigentliche Algorithmus wechselt sich jetzt immer zwischen zwei Phasen ab:

1. Zuordnung: Jeder Datenpunkt x_i wird dem Cluster des nächsten Schwerpunkts zugeordnet.
$$S_i^{(t)} = \{x_j : \|x_j - \mu_i^{(t)}\|^2 \text{ für alle } i = 1, \dots, k\}$$

2. Aktualisieren: Die Schwerpunkte der Cluster werden neu berechnet mit :
$$\mu_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{x_j \in S_i^{(t)}} x_j$$
Wobei $|S_i^{(t)}|$ die Größe des Clusters ist und $\mu_i^{(t+1)}$ der neue Schwerpunkt für den Iterationsschritt $t+1$

Diese zwei Phasen werden so lange wiederholt, bis sich die Zuordnungen nicht mehr ändern. Wenn k und die Anzahl an Dimensionen d fixiert sind, ist die Laufzeit dieses Algorithmus $\mathcal{O}(n^{dk+1})$, wobei n die Anzahl an Objekten ist. k-Means funktioniert recht gut bei Clustern, die ähnlich groß und konvex sind, wenn das „richtige“k vorher bekannt ist. Ausreißer im Datensatz können allerdings dazu führen, dass Cluster nicht gefunden werden.
Es gibt außerdem noch verschiedene Varianten von k-Means. So kann zum Beispiel die Distanzfunktion statt der euklidischen Distanz auch die Manhattan-Norm sein, damit wird auch statt dem Mittelwert der Median der Objekte im Cluster berechnet, diese Variante heißt deshalb auch k-Median. Eine andere Variante ist **fuzzy-c-Means** bei der die Zugehörigkeit zu einem Cluster nicht fest ist, sondern ein Grad an Zugehörigkeit u_{ik} des Objekts x_k mit $k = 1, \dots, N$ zu dem Cluster i mit $i = 1, dots, C$ angegeben wird. N ist dabei die Anzahl an Objekten und C die Anzahl an Clustern. Die verschiedenen Zugehörigkeitsgrade lassen sich also in einer Matrix U zusammenfassen. Die Zielfunktion ist in diesem Fall
$$J(X; U, V) = \sum_{i=1}^{M} \sum_{k=1}^{N} (u_{ik})^m d^2(\mathbf{x}_k, \mathbf{v}_i)$$
Dabei ist $V = [\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_M], \mathbf{v}_i \in \mathbb{R}^n$ ein M-Tupel der Cluster-Prototypen, die bestimmt werden müssen und $m \in (1, \infty)$ ist ein Gewichtungsfaktor der die „Fuzziness“der Cluster bestimmt.
Um das Problem zu beheben, dass die Anzahl an Clustern vorher bekannt sein muss,kann man zum Beispiel Quality Threshold Clustering verwenden bei dem diese Zahl nicht vorher bekannt sein muss.

Quality Threshold Clustering

Die allgemeines Idee hier ist, statt der Anzahl an Clustern den maximalen Durchmesser (Treshold engl. Schwellwert) des Clusters vorzugeben. Der Algorithmus ist in Pseudocode in 2.1 beschrieben. Der Algorithmus läuft dabei folgendermaßen ab: für jeden Datenpunkt wird ein Kandidatencluster gebildet indem so lange Punkte zu dem Kandidatencluster hinzugefügt werden bis der Schwellwert erreicht oder überschritten wird. Wenn für alle Datenpunkte ein Kandidatencluster berechnet wurde, wird das Kandidatencluster mit den meisten Elementen ausgewählt und als Cluster gesetzt. Alle Elemente in diesem Cluster werden aus der Menge der zu clusternden Punkte entfernt und für die restlichen Punkte wird der Algorithmus wiederholt bis alle Punkte auf diese Art geclustert wurden.
Um das Ergebnis zu verbessern, kann man auch noch zusätzlich zum maximalen Durchmesser der Cluster eine Minimalgröße an Elementen fordern.
Der Vorteil von Quality Threshold im Gegensatz zu k-means ist, dass die Anzahl an Clustern nicht von Beginn festehen muss. Außerdem ist die Qualität der Cluster durch die Mindestgröße garantiert, ein weiterer Vorteil ist, dass alle möglichen Cluster berücksichtigt werden, und damit die Qualität weiter verbessert wird. Ein Nachteil dieses Algorithmus ist

die hohe Laufzeit (exponential in der Anzahl an zu clusternden Datenpunkten). Außerdem muss statt der Anzahl an Clustern die Mindestgröße und der Schwellwert bekannt sein.

Algorithm 2.1: Pseudocode für Quality Threshold Clustering

```
    Input: workingList, threshold, minSize
    Output: foundCluster
    // Initialisierung
 1  maxCandidate = workingList
    // main loop
 2  while |maxCandidate | > minSize do
 3      candidateCluster = ∅
        // für jeden Knoten ein Kandidatencluster berechnen
 4      forall v ∈ workingList do
 5          vClust = {v}
 6          forall w ∈ workingList do
 7              if w ≠ v then
 8                  if len(v, w) < threshold then
 9                      vClust = vClust ∪{w}
10          if |vClust | > minSize then
11              candidateCluster = candidateCluster ∪
            // größten Kandidatencluster finden
12          maxCandidate = ∅
13          forall s ∈ candidateCluster do
14              if |s| > |maxCandidate | then
15                  maxCandidate = s
16          foundCluster = foundCluster ∪ maxCandidate
            // alle Knoten in maxCandidate aus workingList löschen
17          forall v ∈ maxCandidate do
18              workingList = workingList ∩v
19  return foundCluster
```

2.2.1 Graph-Clustering

Wenn statt reinen Datenpunkten Knoten in einem Graphen geclustert werden, ist das Problem dabei oft Communitys, also Teilgraphen, die stark miteinander verbunden sind, zu finden. Graphclustering eines Graphen $G = (V, E)$ ist die Clusterung (Partitionierung) von V, so dass eine Zielfunktion optimiert wird. Diese Zielfunktion kann zum Beispiel das Gewicht der Kanten, die zwischen den Clustern verlaufen, also der Schnitt zwischen diesen Clustern, die Modularität oder noch andere Metriken sein.

Louvain-Clustering

Louvain-Clustering ist hierarchisches Graphclusteringverfahren, bei dem heuristisch die Modularität optimiert wird wie in [NG04] beschrieben.
Gegeben ein gewichteter Graph $G = (V, E, \omega)$. Dann istdie Modularität ein Wert, der den

Bruchteil der Kanten, die innerhalb der Cluster sind, minus den erwarteten Bruchteil der Kanten, falls die Kanten zufällig in die Cluster verteilt wären, angibt.

$$Q = \frac{1}{2m} \sum_{ij} [\omega(ij) - \frac{k_i k_j}{2m}] \delta(\omega_i, \omega_j)$$

Dabei ist ω das Kantengewicht, zwischen den Knoten auf der Kante $\{i, j\} \in E$, k_i, k_j ist jeweils die Summe der Kantengewichte an den Knoten i bzw. j, $m = \sum_{e \in E} c(e)$ ist die Summe aller Kantengewichte im Graphen, δ ist eine einfache Deltafunktion, $\delta_{ij} = 0$ falls $i \neq j$ und $\delta_{ij} = 1$ falls $i = j$. Der Wert der Modularität ist dabei immer $Q \in [-1, 1]$, eine hohe Modularität bedeutet dabei wenige Kanten, die zwischen den Clustern verlaufen also eine gute Clusterung, eine niedrige Modularität zeigt, dass wenige viele zwischen den Clustern verlaufen die Clusterug also eher schlecht ist.
Da es NP-schwer ist die Modularität zu optimieren werden dafür Heuristiken verwendet. Die Louvain-Methode ist eine Graphclustering-Methode die diese Zielfunktion optimiert. Initial ist dabei jeder Knoten in seinem eigenen Cluster.
Danach wird versucht die Zielfunktion zu verbessern indem sich immer zwei Phasen abwechseln: Für jeden Knoten $i \in V$ wird die Veränderung der Zielfunktion für jeden Nachbarknoten $j \in N(i)$ berechnet, wenn i in den Cluster des Nachbarknoten $j \in N(i)$ wechselt. Nachdem dies für alle Nachbarn berechnet wurde wechselt i in den Cluster desjenigen Nachbarn, bei dem die Zielfunktion am meisten verbessert wird. Falls keine Verbesserung möglich ist, bleibt i in seinem eigenen Cluster. Dies wird solange für alle Knoten $v \in V$ wiederholt, bis keine Verbesserungen mehr auftreten, dabei ist die Abarbeitungsreihenfolge der Knoten zufällig.
Dann beginnt Phase 2: Dabei werden alle Knoten in einem Cluster zu einem Knoten kontrahiert. Dabei wird das Kantengewicht zwischen den neuen Knoten so angepasst, dass das Gewicht der Kante zwischen zwei neu kontrahierten Knoten, die Summe der Kantengewichte der Kanten, die zwischen den beiden Clustern verliefen ist. Außerdem werden alle Kanten innerhalb des Clusters zu einer Schleife zusammengefasst, diese Schleife hat als Kantengewicht die Summe der Kantengewichte der Kanten innerhalb des Clusters. Damit bleibt die Modularität erhalten. Diese zwei Phasen wiederholen sich nun solange bis nur noch ein Knoten übrig bleibt. Die Laufzeit des Louvain-Algorithmus ist in empirischen Beobachtungen bei $\mathcal{O}(n \log n)$.
Auch wenn Louvain-Clustering mit der Modularität als Zielfunktion vorgestellt wurde und meistens so verwendet wird, kann dieses Verfahren aber auch mit anderen Zielfunktionen durchgeführt werden.

3. Windfarmodellierung

Meine Modellierung basiert im Wesentlichen auf [LRWW17]. Für diese Problemstellung kann eine Windfarm als Graph $G = (V, E)$ modelliert werden. Die Knotenmenge V lässt sich aufteilen in die Menge der WEA V_T und die Menge der Umspannstationen V_S, dabei gilt $V = V_T \cup V_S$. Bevor ich damit eine Windfarm modellieren kann, benötige ich jedoch noch folgende weiteren Komponenten für die Modellierung der Kabel:
Die Menge K an Kabeltypen, eine Funktion $\text{cap} : K \to \mathbb{R}_{\geq 0} \cup \{\infty\}$, die jedem Kabeltyp eine maximale Kapazität $cap(\kappa)$ zuordnet. Sowie die Funktion, $c_{\text{cab}}(\kappa) : K \to \mathbb{R}_{\geq 0} \cup \{\infty\}$, für die Kosten $c_{\text{cab}}(\kappa)$ ein Kabel vom Typ κ eine Längeneinheit zu verlegen. Für $\kappa_1, \kappa_2 \in K$ gilt $\kappa_1 < \kappa_2$ genau dann wenn $\text{cap}(\kappa_1) < \text{cap}(\kappa_2)$ und $\text{cap}(\kappa_1) < \text{cap}(\kappa_2)$. o.B.d.A. ist $<$ eine strenge Totalordnung auf K. Eine strenge Totalordung ist eine Relation $<$ auf einer Menge M so dass $x < y \wedge y < z \Rightarrow x < z$ sowie entweder $x < y$ oder $x = y$ oder $y < x$ für alle $x, y, z \in M$ gilt. Außerdem gibt es zwei Spezialkabel $\kappa_0, \kappa_\infty \in K$ mit $\text{cap}(\kappa_0) = c_{\text{cap}}(\kappa_0) = 0$ und $\text{cap}(\kappa_\infty) = c_{\text{cap}}(\kappa_{infty}) = \infty$. Das erste Spezialkabel, wird benötigt um Kanten ohne Kabel zu modellieren, das zweite Spezialkabel damit jede Instanz eine gültige Lösung hat, eventuell mit unendlichen Kosten.
Für die Modellierung des Kabellayouts der gesamten Windfarm, im folgenden FFP (Full Farm Problem) genannt, verwende ich noch folgende zusätzliche Parameter:

- Die maximale Anzahl WEA pro Stromkreis n_C, dieser Wert kommt vom Nennstrom des dicksten erhältlichen Kabels.
- Für jede WEA $u \in V_T$ die Nennleistung $p_u \in \mathbb{R}_{\geq 0}$, die sie liefern kann.
- Für jede Umspannstation $s \in V_S$ die maximale Kapazität $d_v \in \mathbb{R}_{\geq 0}$
- Die Länge $\text{len}(u, v)$ einer Kante $(u, v) \in E$

Eine Instanz des Problems besteht damit aus dem gewichteten bidirektionalen Graphen $G = (V, E, \text{len})$, wobei die Kantenmenge E die möglichen Verbindungen in der Windfarm darstellt. Die Lösung einer solchen Instanz des FFP ist ein Paar (κ, f), dabei ist $\kappa : E \to K$ eine Funktion, die jeder Kante ein Kabel zuordnet und f ein Fluss auf G, der Flusserhaltung und Kantenkapazitäten erfüllt wie in 3.4 beschrieben. Dabei sind die Umspannstationen

Senken, mit der Kapazität n_S und die WEA Quellen, die jeweils eine Einheit Fluss generieren.

$$\sum_{u \in V} f_{net}(u) = 0 \tag{3.1}$$

$$f_{net}(u) = -1 \qquad u \in V_T \tag{3.2}$$

$$0 \leq f_{net}(u) \leq n_S \qquad u \in V_S \tag{3.3}$$

$$f(e) \leq cap(\kappa(e)) \qquad e \in E \tag{3.4}$$

Ein Paar κ, f, das diese Bedingungen erfüllt, heißt gültig. Das Flussproblem zum FFP ist also $\mathcal{N} = (G, K, \text{cap}(\cdot), c_{\text{cab}}, \mathbf{p}, \mathbf{d})$, dabei ist $\mathbf{p} = \{p_u\}_{u \in V_t}$ der Versorgungssvektor und $\mathbf{d} = \{d_v\}_{v \in V_S}$ der Bedarfsvektor. Die Gesamtkosten einer gültigen Lösung κ, f berechnen sich mit

$$c(\kappa, f) = \sum_{e \in E} c_{cab}(\kappa(e)) \cdot len(e)$$

Mein Ziel ist es die Gesamtkosten zu minimieren, die minimalen Kosten einer solchen Lösung heißen OPT($\mathcal{N}$). Es ist leicht zu sehen, dass gegeben ein Fluss f auf G eine Zuordnung der Kabel an Kanten κ so dass die Kosten minimal sind, einfach zu finden ist. Die Kosten für eine Kante $e \in E$ sind: $c_e(x) = \min\{c_{\text{cab}}(\kappa) | \kappa \in K, x \leq \text{cap}(\kappa)\} \cdot \text{len}(e)$. Und damit kann das Problem als ein Min-Cost-Flow-Problem geschrieben werden: $N = (G, c_e(\cdot))$. Analog zu [BVMO11] kann das Problem in noch in zwei Unterprobleme aufgeteilt werden:

Substation Problem (SP)

Das Substation Problem betrachtet das Kabellayout eines Sammelsystem G^j einer Umspannstation und aller an diese Umspannstation über unterschiedliche Stromkreise angeschlossene WEA innerhalb der Windfarm G. Das Problem besteht darin die Kosten der Verkabelung dieses Sammelsystems zu minimieren. Das `SP` ist ein Unterproblem des FFP, da die Sammelanlage G^j ein Teilgraph der Windfarm G ist. Die Sammelsystem G^j lässt sich definieren als $G^j := G - V \backslash (\{j\} \cup V_T^j)$, dabei ist V_T^j die Menge der WEA, die zur Sammelanlage der Umspannstation $j \in V_s$ gehören. Außerdem ist die Menge der Umspannstationen beschränkt auf $V_S = \{j\}$ mit $|V_s| = 1$ und die Menge der WEA V_t^j für das Sammelsystem G^j der Umspannstation $j \in V_s$ ist fixiert. Die Knotenmenge $V^j \subseteq V$ ist damit also definiert als $V^j = V_T^j \cup \{j\}$ und die Kantenmenge ist $E^j \subseteq V^j \times V^j$ mit $E^j \subseteq E$. Die Anzahl an Stromkreisen in disem Sammelsystem ist allerdings nicht bekannt, aber nach oben beschränkt durch $|V_T^j|$. Damit lässt sich die Umspannstationenkapazität von 3.4 reduzieren auf $f_{\text{net}}(j) = d_j = \sum_{u \in V_t^j} p(u)$. Das Flussproblem für das `SP` ist also beschrieben durch $\mathcal{N}_{\texttt{SP}}(j) = (G^j, c_{textcab}(\cdot), \mathbf{p}, \mathbf{d})$. Der Wert einer Lösung f^* auf E^j, die $c(f^*)$ minimiert, heißt dabei OPT($\mathcal{N}_{\texttt{SP}}(j)$).

Circuit Problem (CP)

Das Circuit Problem ist das kleinste Unterproblem in der Hierarchie der Windfarmlayoutprobleme, es betriftt nur einen einzelnen Stromkreis, also einen verbundenen Komponenten, der mit einer Umspannstation verbunden ist. Das Ziel ist dabei wieder eine Lösung zu finden, die die Kabelkosten minimiert. Das `CP` ist also ein Unterproblem des `SP`, da ein Stromkreis $G^{j,i}$ ein Teilgraph eines Sammelsystems G^j mit der Umspannstation $j \in V_S$ und dem Stromkreis $i \in \mathbb{N}$ ist. Dabei bleibt die Anzahl an Umspannstationen bei $|V_S| = 1$, technisch gesehen, wird aber nur ein Anschluss an der Umspannstation j betrachtet. Gegeben ist also die Menge der WEA $V_T^{j,i}$ im Stromkreis j in dem Sammelsystem G^j an der Umspannstation j. Die Knotenmenge $V^{j,i} \subseteq V^j$ ist damit $V^{j,i} = V_T^{j,i} \cup \{j\}$ und die Kantenmenge $E^{j,i} \subseteq E^j$ ist $E^{j,i} = V^{j,i} \times V^{j,i}$. Der für den Stromkreis i betrachtete Graph ist also $G^{j,i} = (V^{j,i}, E^{j,i})$. Außerdem lässt sich damit die Gleichung 3.4 beschränken auf

$f_{\text{net}}(j) = d_j^i = \sum_{u \in V_T^{j,i}} p_u$, wobei d_j^i der Bedarf für den Stromkreis i an der Umspannstation j ist. Das Flussproblem für das `CP` ist damit $\mathcal{N}_{\texttt{CP}}(j,i) = (G^j, c_{textcab}(\cdot), \mathbf{p}, \mathbf{d})$. Der Wert der Lösung f^*, die $c(f^*)$ minimiert heißt dabei $\text{OPT}(\mathcal{N}_{\texttt{CP}}(j,i))$.

Innerhalb einer (geplanten) Windfarm werden normalerweise WEA des gleichen Typs verwendet, damit ist auch die Nennleistung aller WEA die gleiche und p_u kann deshalb auf 1 normiert werden für alle $u \in V_T$. Um die Ergebnisse besser vergleichen zu können, sind auch die Umspannstationen alle vom gleichen Typ und haben die Kapazität n_S. Damit ist das Flussproblem des `FFP` vereinfacht zu $\mathcal{N} = (G, K, \text{cap}(\cdot), c_{\text{cab}}(\cdot), n_S)$

4. Untersuchte Clusteringalgorithmen

Ich habe mehrere Varianten eines „Mover“-Algorithmuses untersucht, sowie zum Vergleich auch den Multilevel-Quality-Threshold-Algorithmus von [DO11].

4.1 Moveralgorithmus

Dieser Algorithmus basiert auf [BHSW15]. Lokale Move Algorithmen sind schon lange für Clusteringprobleme verwendet worden, so basiert die Louvain-Methode zum Beispiel darauf.
Die Mover-Algorithmen sind Bergsteiger-Algorithmen, die ausgehend von einer gültigen Lösung verbessernde Lösungen suchen. Die Idee ist dabei, dass für eine WEA $t \in V_T$ betrachtet wird, wie sich die Kostenfunktion verändert, wenn t mit einer speziellen Move-Operation einen neuen Elter erhält. Wenn es dadurch eine verbessernde Lösung gibt, wird t mit dem Knoten $v \in V$ verbunden, der die größte Verbesserung ergibt und t wechselt in das Cluster von v. Dies wird immer in Runden für alle WEA $v_t \in V_T$ durchgeführt bis keine Verbesserungen mehr auftreten, oder die Schleife mehr als n-mal durchlaufen wurde mit $n \in \mathbb{N}_{>0}$.
Der generelle Algorithmus wird in 4.1 beschrieben. Dabei verwende ich folgende Funktionen: $\textsc{shuffle}(V_T)$, permutiert die WEA-Menge, `cluster`(v) gibt den Cluster, zu dem der Knoten $v \in V$ gehört zurück, `merge`(c_1, c_2) verschmilzt die beiden Cluster c_1 und c_2. Zunächst wird der Algorithmus initialisiert mit einer Initiallösung, sowie werden die Cluster initialisiert, dies bedeutet dass alle Knoten $v \in V$ in einem eigenen Cluster sind. Außerdem wird der `counter` auf 0 gesetzt, und die Kosten `cost` der Initiallösung berechnet. Da die Schleife immer das aktuelle `cost` mit den Kosten `costOld` der letzten Lösung vergleicht, ob sich die Lösung verbessert hat, wird `costOld` auf `cost` +1 gesetzt.
In der eigentlichen Schleife wird zunächst `costOld` auf `cost` gesetzt. Und eine weitere Variable `costTemp` mit `cost` gesetzt. Die WEA werden dann immer in einer zufälligen Reihenfolge bearbeitet. Für jede WEA $t \in V_T$ werden alle adjazenten Knoten n betrachtet. Falls die Move-Operation, die n zum neuen Elter von t ändern würde, einen gültigen Fluss F' erzeugt und $c(F') <$ `costTemp` wird F' gespeichert und `costTemp` auf $c(F')$ gesetzt. Wenn t und n in unterschiedlichen Clustern waren, werden die beiden Cluster dann verschmolzen. Die beste Lösung wird als neuer Fluss für die nächste betrachtete WEA verwendet.

Bei allen Mover-Varianten wird der Algorithmus mehrfach ausgeführt um die unterschiedlichen Lösungen, die durch die zufällige Abarbeitungsreihenfolge zustande kommen, zu berücksichtigen und die Beste zu finden. Dies ist in 4.2 beschrieben.

Algorithm 4.1: Pseudocode der das generelle Vorgehen der Mover-Algorithmen beschreibt

```
    Input: G = (V, E), n_S, n_C
    Output: flow, cost
    // Initialisierung
 1  G_Kabellayout = ordne initial WEA zu Umspannstationen
 2  counter = 0
 3  forall v ∈ V do
 4      initialisiere Cluster
 5  cost = c(κ, f) = Σ_{e∈E} c_cab(κ(e)) · len(e)
 6  costOld = cost + 1;
    // Main loop
 7  while cost < costOld ∨ counter < stop do
 8      costOld = cost
 9      costTemp = cost
10      SHUFFLE(V_T)
11      forall v ∈ V_T do
12          clustTemp = CLUSTER(v)
13          forall n ∈ Adjazenzliste(v) do
14              F' = MOVER(v, n, PARENT(v))
15              if F' zulässig then
16                  if F'.cost < cost then
17                      costTemp = F'.cost
18                      clustTemp = CLUSTER(n)
19                      flow = F'.flow
20          if clustTemp ≠ CLUSTER(v) then
21              MERGE(clustTemp, CLUSTER(v))
22      counter ++
23  return (G_Kabellayout,cost)
```

Die drei unterschiedlichen Varianten des Mover-Algorithmusses werden in der Abbildung 4.1 illustriert.

4.1.1 Standard-Mover

In dieser Variante werden die WEA immer mitsamt ihrer Kinder bewegt. Die WEA u erhält also einen neuen Elter, aber u-s Kinder bleiben so erhalten. Diese Operation ist als Pseudocode in 4.3 beschrieben. Dabei ist isForest eine Funktion, die den Fluss darauf überprüft ob er ein Wald ist, also keine Kreise vorhanden sind, und der Fluss immer in Richtung Wurzel fließt, isTurbine überprüft ob ein Knoten eine WEA ist, parent gibt den Elter eines Knoten zurück.
Die Operation läuft so ab, dass zunächst der Fluss auf der Kante $(u, oldParent)$ in der Variable flowDiff gespeichert wird, der Fluss auf dieser Kante wird dann auf 0 gesetzt wird. Danach wird der Fluss auf der Kante $(u, newParent)$ auf flowDiff gesetzt. Diese neue Topologie wird dann zunächst darauf überprüft, ob dadurch Kreise entstanden sind, bzw. dass der Fluss nicht in Richtung Wurzel fließt. Falls beides nicht der Fall ist, wird nun zunächst der Fluss auf allen Kanten e des Astes, der von oldParent zur Umspannstation geht, angepasst und dabei auch die Kosten aktualisiert. Dabei wird immer flowDiff von $f(e)$ abgezogen, sowie die alten Kosten der Kante von cost abgezogen und die neuen Kosten der

Algorithm 4.2: Pseudocode der beschreibt wie oft der Algorithmus wiederholt wird

```
    Input: G = (V, E)
    Output: flow
    // Initialisierung
 1  counter = 0
 2  while counter < 30 do
 3  |   costTemp = MOVER ()
 4  |   if (costTemp < cost
 5  |   flow = MOVER ()
 6  |   cost = costTemp
 7  |   counter = 0
 8  |   else
 9  |   counter ++
10  return flow
```

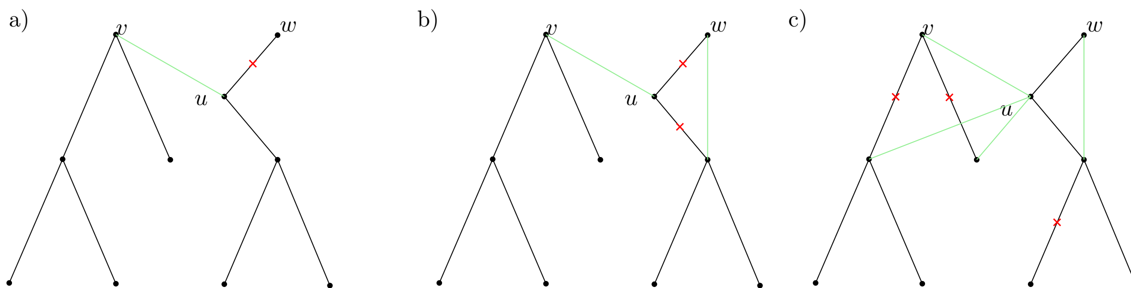

Abbildung 4.1: Die 3 Mover Varianten im Vergleich. a) ist der Standard Mover, b) der Single-Mover, c) der Adoptionsmover. Die grünen Kanten sind jeweils die Kanten, die neu hinzukommen, die roten Kreuze markieren Kanten, die gelöscht werden. Der Knoten u ist der Knoten, der bewegt wird; v der neue Elter von u; w der alte Elter von u.

Kante aufaddiert. Danach wird der Ast von newParent zur Umspannstation betrachtet. Hier wird flowDiff auf $f(e)$ addiert. Dieser neue Fluss wird dann darauf überprüft, ob er die Kapazitäten noch einhält. Falls dies der Fall ist, werden die Kosten aktualisiert.
Am Schluss wird der neue Fluss und die dazugehörigen Kosten zurückgegeben.

4.1.2 Single-Mover

Dies ist eine Verbesserung des obigen Algorithmus. Die Move-Operation dieser Variante ist in 4.4 beschrieben. Die Funktion CHILDREN dort ist eine Funktion die eine Liste der Kinder des Knotens zurückgibt. Der Unterschied zwischen der Standard-Move-Operation und der Singlemove-Operation besteht darin, dass der Knoten u ohne seine Kinder verschoben wird. Die Kinder von u werden mit ihrem "Großelter" verbunden. Entsprechend muss dann f angepasst werden so dass der Fluss auf den Kanten zwischen den Kindern und u 0 ist und dafür der Fluss, der davor auf dieser Kante war, auf der Kante zwischen dem Kind und dem Elter von u.
In der lokalen Suche wird zunächst mit dem Standardmoveralgorithmus gesucht, bis dies keine Verbesserungen mehr liefert. Dann wird für alle WEA geprüft ob eine Single-Move-Operation zu einer Verbesserung führt, wenn ja wird die beste solche als neuer Fluss gesetzt. Falls es in dieser Runde eine Verbesserung gab, wird wieder so lange mit dem Standardmoveralgorithmus gesucht bis dieser keine Verbesserungen mehr findet. Falls

Algorithm 4.3: Pseudocode, der die Move-Operation im Standard Fall beschreibt

```
   Input: flowOld, cost, u, oldParent, newParent, n_S, n_C
   Output: flowNew, cost
   // Initialisierung
 1 flowNew = flowOld
 2 flowDiff = flowOld[u,oldParent]
 3 flowNew[u,oldParent] = 0
 4 cost = cost- c_cab(flowDiff ) · len(u,oldParent )
 5 flowNew[u,newParent] = flowDiff
 6 cost = cost + c_cab(flowDiff ) · len(u,newParent ) if !isForest(flowNew) then
 7     return none
   // flowDiff vom alten Pfad abziehen
 8 it = oldParent
 9 while isTurbine(it) do
10     flowNew[it, parent(it)] = flowOld[it, parent(it)] - flowDiff
11     cost = cost −(c_cab(flowOld[it, parent(it)]) + c_cab(flowNew[it,
         parent(it)])) · len(it, parent(it))
12     it = parent(it)
   // flowDiff auf neuen Pfad aufaddieren
13 it = newParent
14 while isTurbine(i) do
15     flowNew[it, parent(it)] = flowOld[it, parent(it)] + flowDiff
16     if isTurbine(parent(it)) ∧ flowNew[it, parent(it)] ≥ n_S then
17         return None
18     if !isTurbine(parent(it)) ∧ flowNew[it, parent(it)] ≥ n_C then
19         return None
20     cost = cost −(c_cab(flowOld[it, parent(it)]) + c_cab(flowNew[it,
         parent(it)])) · len(it, parent(it))
21 return flowNew, cost
```

der Single-Move auch keine Verbesserungen gefunden hat, oder insgesamt mehr als eine spezifizierte Anzahl Runden verstrichen sind, bricht die Suche ab.

4.1.3 Adoptionsmover

Auch diese Variante ist eine Verbesserung des Standardmovers. Der Pseudocode der entsprechenden Move-Operation ist 4.5. Die Idee hierbei ist, ähnlich wie beim Singlemove den Knoten ohne seine Kinder zu bewegen, aber im Gegensatz zum Singlemove wird eine Teilmenge der Kinder des neuen Elter von u adoptiert. Für die lokale Suche bedeutet das, dass grundsätzlich wie auch beim Single-Move mit dem Standardmoveralgorithmus gesucht wird bis dieser keine Verbesserungen findet, und erst wenn dieser keine Verbesserungen mehr findet, wird der Adoptionsmover verwendet. Dabei werden für jede WEA t alle dazu adjazenten Knoten n betrachtet. Um die Suche zu beschleunigen, werden nur diejenigen Kinder von n betrachtet, die näher an t als an n sind, Von diesen Kindern werden alle Teilmengen getestet. Falls damit eine Verbesserung gefunden wurde, wird wieder mit dem Standardmoveralgorithmus gesucht, bis dieser keine Verbesserungen findet. Die Suche wird abgebrochen, wenn entweder beide Move-Operationen keine verbessernde Lösung mehr finden, oder wenn insgesamt mehr als eine vorher spezifizierte Anzahl Runden durchlaufen wurde.

Algorithm 4.4: Pseudocode, der die Single-Move-Operation beschreibt

```
   Input: flowOld, cost, u, oldParent, newParent, n_S, n_C
   Output: flowNew, cost
   // Initialisierung
 1 flowNew = flowOld
 2 flowNew[u,oldParent] = 0
 3 cost = cost- c_cab(flowOld[u, oldParent]) · len(u,oldParent )
 4 flowNew[u,newParent] = 1
 5 cost = cost + c_cab(1) · len(u,newParent ) if !isForest(flowNew) then
 6     return none
   // 1 vom alten Pfad abziehen
 7 it = oldParent
 8 while isTurbine(it) do
 9     flowNew[it, parent(it)] = flowOld[it, parent(it)] - 1
10     cost = cost −(c_cab(flowOld[it, parent(it)]) + c_cab(flowNew[it,
        parent(it)])) · len(it, parent(it))
11     it = parent(it)
   // alle Kinder von u werden von oldParent adoptiert
12 forall c ∈ children(u) do
13     flowNew[c,u] = 0
14     flowNew[c,oldParent]= flowOld[c, u]
15     cost = c_cab(flowOldc, u) · (len(c,oldParent ) − len(c, u))
   // 1 auf neuen Pfad aufaddieren
16 it = newParent
17 while isTurbine(i) do
18     flowNew[it, parent(it)] = flowOld[it, parent(it)] + 1
19     if isTurbine(parent(it)) ∧ flowNew[it, parent(it)] ≥ n_S then
20         return None
21     if !isTurbine(parent(it)) ∧ flowNew[it, parent(it)] ≥ n_C then
22         return None
23     cost = cost −(c_cab(flowOld[it, parent(it)]) + c_cab(flowNew[it,
        parent(it)])) · len(it, parent(it))
24 return flowNew, cost
```

4.1.4 Multilevel Quality Threshold

Dieser Algorithmus basiert auf [DO11]. Die generelle Idee dabei ist mehrfach Quality Threshold Clustering durchzuführen: Zunächst werden die WEA mit einem initialen Threshold geclustert. Die minimale Größe eines Clusters beträgt 4 WEA. Für jedes gefundene Cluster wird ein Repräsentant bestimmt, als die WEA mit dem kleinsten Abstand zur nächsten Umspannstation. Im nächsten Schritt wird der Threshold verdoppelt und alle Repräsentanten sowie die davor noch nicht geclusterten WEA damit geclustert. Dies wird so lange wiederholt, bis der größte Cluster n_C Elemente hat. Als finaler Schritt werden alle noch nicht geclusterten WEA sowie die Repräsentanten mit ihrer nächsten Umspannstation verbunden. Der Pseudocode davon ist in 4.6 zu finden. Dabei ist closestSubstation eine Funktion, die für jeden Knoten, die Umspannstation mit dem kleinsten Abstand zurückgibt; representatives ist eine Abbildung, die Repräsentanten auf Cluster abbildet; minDistance ist eine Funktion, die die kürzeste Länge einer Kante in E zurückgibt; qualityThresholdClustering(workingList, threshold) ist schließlich eine Funktion, die Quality Threshold Clustering auf den Knoten in workingList mit threshold durchführt,

wie in 2.1 beschrieben. Für ein möglichst gutes Ergebnis teste ich bei jedem Graphen verschiedene initiale Thresholds indem der initialMultiplyer zwischen 1 und 30 variiert wird.

4.1.4.1 Multilevel Quality Threshold mit Kostenfunktion

Dies ist eine Verbesserung des obigen Algorithmus. Die Idee ist, statt einfach nur mit dem euklidischen Abstand zu clustern, das benötigte Kabel und die Kosten dafür zu schätzen. Für diese Schätzung wird beim Berechnen des Kandidatencluster um den Repräsentanten v immer die Größe des Clusters, das v repräsentiert, als Größe des benötigten Kabels verwendet. Als weitere Verbesserung werden die WEA auch nicht sternförmig mit ihren Repräsentanten verbunden, sondern es werden minimale Spannbäume der Cluster berechnet mit dem Algorithmus von Prim. Für die Berechnung des Spannbaums werden wiederum die Kosten abgeschätzt. Dafür wird als Kantengewicht jeder Kante (u, v) $\text{len}(u, v) \times c_{\text{cab}}(clusterSize(u))$ gesetzt. Der Pseudocode dieses Algorithmusses ist in 4.7 zu finden. Die Funktionen, die dabei zusätzlich zu oben verwendet werden sind: QUALITYTHRESHOLDCLUSTERINGCOST die ist wie oben beschrieben eine Funktion, die Quality Threshold Clustering mit der einer Schätzung der Kostenfunktion durchführt. Dabei wird beim Aufbauen des Kandidatenclusters c_v um v, für die Überprüfung, ob der Knoten w noch aufgenommen werden kann, immer die Größe des Clusters, das v repräsentiert als Fluss auf der Kante (v, w) geschätzt.
PRIM berechnet einen Minimalen Spannbaum mit Hilfe Prims Algorithmus; NEARESTVALIDNEIGHBOUR findet den nächsten Nachbarn, an den dieser Knoten angeschlossen werden kann, ohne die Kapazitätsbedingungen zu verletzen.
Auch in diesem Fall teste ich wieder bei jedem Graphen verschiedene initiale Thresholds in dem der initialMultiplyer zwischen 1 und 30 variiert wird.

4.2 Fallstudien

4.2.1 Daten

Ich habe die Benchmark-Graphen von [LRWW17] verwendet. Dies sind generierte Windfarmen in verschiedenen Größen. Die Windfarmen haben dabei immer eine elliptische Form, die Form der Ellipse lässt sich dabei durch ihr Seitenverhältnis $\beta \in (0, 1]$ beschreiben. Die WEA und Umspannstationen sind jeweils mit Poisson-Verteilung platziert, so dass sie einen minimalen Abstand zueinander haben. Die so generierten Windfarmen lassen sich dabei durch verschiedene Parameter charakterisieren: $|V_T|$, $|V_S|$ sowie die "Tightness" der Umspannstationenkapazität $\gamma = \frac{|V_T|}{n_S \cdot |V_S|}$. Die Kantenmenge E der generierten Graphen sind dabei mögliche Kabelverbindungen. Dabei haben alle Knoten Kanten zu ihren k nächsten Nachbarn, außerdem werden noch zusätzliche Kanten E' eingefügt für Knoten, die noch keine Kante in E haben:

$$E' := \{\{u, w\} \notin E : \text{len}(u, v) + \text{len}(v, w) > \epsilon \cdot (u, w)\}$$

mit $\{u, v\}, \{v, w\} \in E$. Die Graphen lassen sich damit in 5 Mengen $\mathcal{N}_1 - \mathcal{N}_5$ unterteilen. Dabei haben die Graphen in den Mengen $\mathcal{N}_1 - \mathcal{N}_4$ eine eingeschränkte Kantenmenge E wie oben beschrieben, und die Graphen in $\mathcal{N}_5$ sind vollständige Graphen. Dabei habe ich mich hauptsächlich auf die Mengen $\mathcal{N}_1 - \mathcal{N}_4$ beschränkt, da diese auf einer begrenzten Kantenmenge arbeiten. Die Details der einzelnen Benchmarkmengen sind in 4.1 zu finden.

Außerdem verwende ich die Daten aus Berzan et. al. [BVMO11] für die Kosten und Kapazitäten der Kabel, sowie der Umspannstationenkapazität, die dort von Experten genannt wurden. Damit ist die Umspannstationenkapazität n_S 42 und für die Kabel gelten die Werte aus 4.2:

4.2.2 Durchführung

Außer mit den verschiedenen Varianten des Mover-Algorithmuses habe ich alle Graphen, auch mit den beiden Multilevel-Quality-Threshold- Varianten sowie einem modifizierten MST verglichen. Den modifizierten MST habe ich dabei folgendermaßen berechnet: Zunächst habe ich einen modifizierten Graphen $G' = (V', E')$ konstruiert. Dieser hat eine Dummy-Wurzel r mit $V' = V \cup r$ und zusätzliche Kanten so dass $E' = E \cup \bigcup_{u \in V_S} (u, r)$. Dann wird auf G' ein MST mit dem Algorithmus von Prim berechnet, mit r als dedizierter Wurzel und dem Kantengewicht auf allen Kanten $e \in E$ als $\text{len}(e)$. Aus diesem MST lösche ich dann r und alle inzidenten Kanten dazu und erhalte damit einen Spannwald. Von diesem berechne ich dann die Kosten.
Als zusätzliche Varianten meiner 3 Moveralgorithmen habe ich die Initiallösung variiert. Diese Varianten sind:

- Eine zufällige gleichmäßige Aufteilung der WEA an die Umspannstationen. Dies bedeutet, dass die WEA sternförmig mit der Umspannstation verbunden sind und die Anzahl an WEA pro Umspannstation unterscheidet sich maximal um 1. Da es dabei möglich ist, dass eine WEA $t \in V_T$ zur Umspannstation $s \in V_S$ zugeordnet wird, obwohl die Kante $\{v, s\}$ nicht im Inputgraphen ist, wird dann eine Kante eingefügt mit dem Kantengewicht INTMAX.
- Ein aufspannender Wald wie oben beschrieben.

Die Algorithmen wurden in C++14 mit dem gcc 7.1.1 20170528 implementiert unter Verwendung des Open Graph Drawing Frameworks in der Version snapshot-2017-02-16 [CGJ+12].

Generell kann man erkennen, dass die Adoptionsvariante, die besten Ergebnisse liefert und das Ergebnis von der Qualität der initialen Lösung abhängt.

Über alle Instanzen ergeben sich im Vergleich mit dem Mixed Integer Linear Programming-Algorithmus (MILP) aus [LRWW17] dabei die Werte wie in der Tabelle 4.2.2 dargestellt. Im besten Fall konnte mit diesen Algorithmen ein Kabellayout berechnet werden bei dem die Kosten nur 96,5% der Kosten des MILP sind, im schlechtesten Fall allerdings auch ein Kabellayout bei dem die Kosten 174,5% der Kosten des MILP betragen. Die beiden Varianten des Movers mit Adoption bzw. der Single-Move sind dabei ähnlich gut. Beim Adoptionsmoveralgorithmus mit der Initialisierung basierend auf einem Minimalen Spannbaum ist der Median bei 1,001 und 46,16% der Lösungen sind besser als die Lösungen des MILP, beim Single-Move-Algorithmus mit der Initialiserung basierend auf dem MST ist der Median ebenfalls bei 1,001 und 44,67% der Lösungen sind besser als die des MILP; die schlechteste Movervariante ist der Standardmover mit der Standardinitialiserung mit einem Median von 1,038.
Die Multilevel Quality Threshold Algorithmen sind dabei schlechter als die Moveralgorithmen. Dabei ist der einfache Multilevel Quality Threshold Clustering Algorithmus mit einem Median von 1,683 am schlechtesten, aber auch die kostenbasierte Variante hat nur einen Median von 1,103. Dabei ist das Ergebnis in allen Fällen schlechter als das des MILP. Auch der Minimale Spannbaum liefert mit einem Median von 1,188 schlechtere Ergebnisse als die Moveralgorithmen und insgesamt sind dort nur in 0,48% der Fälle die Ergebnisse besser als die des MILP.

Über die Anzahl an Turbinen betrachtet kann man erkennen, dass Instanzen die Moveralgorithmen und insbesondere der Adoptionsmoveralgorithmus mit MST-Initialiserung sehr gute Ergebnisse liefert. Die Ergebnisse sind nie viel schlechter als die des MILP. In 4.2 und in 4.3 sind die Plots meiner Daten im Vergleich zum MILP aufgetragen, jeweils in allen 3 Movervarianten, in **??** die relativen Kosten der Multilevel-Quality-Threshold-Algorithmen sowie des MST im Vergleich zum MILP.

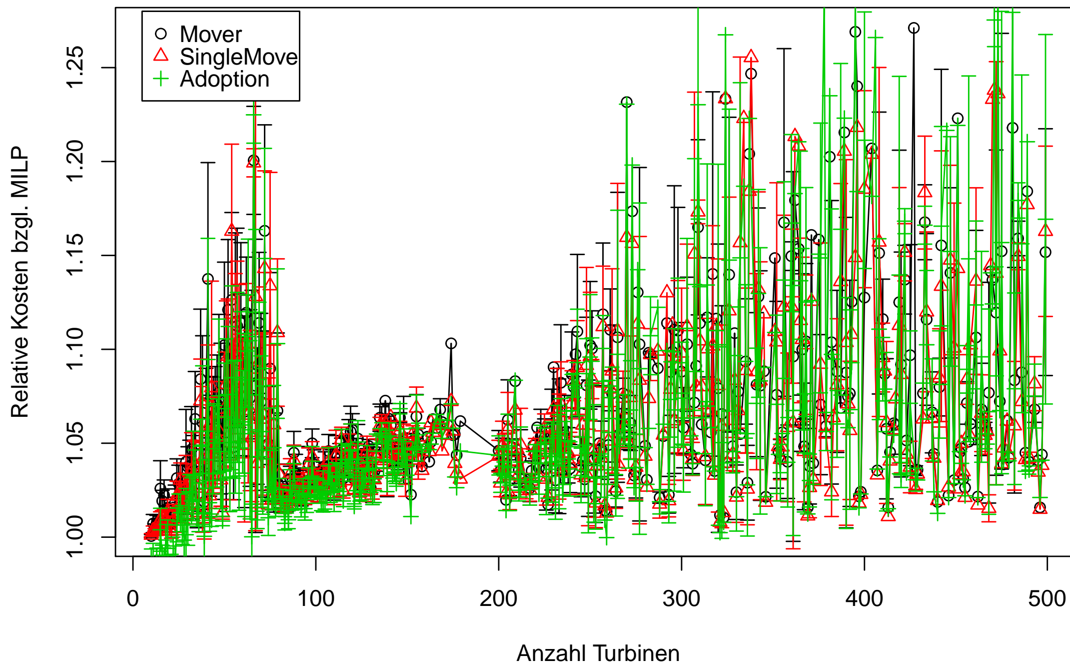

Abbildung 4.2: Ein Vergleich der drei Movervarianten bei der Standardinitialiserung. Schwarz ist der Standardmover, Rot der Single-Move-Algorithmus, Grün der Adoptionsmover. Die Kosten sind dabei immmer im Verhältnis zu den Kosten der Lösung des MILP aufgetragen, als der Mittelwert und die Fehlerbalken sind der Standardfehler

Im Vergleich zum MILP ist mein Algorithmus dabei besonders für Instanzen unter 200 WEA deutlich schneller. Alle im folgenden genannten Zeiten wurden auf einem AMD Opteron™Processor 6172 ermittelt. Für eine kleine Instanz mit 10 WEA und einer Umspannstation benötigt der Standardmover sowohl mit Standardinitialiserung als auch bei der MST-Initialiserung nur 30 ms, der Single-Move-Algorithmus benötigt für die gleiche Instanz 80ms bzw. 70ms, der Adoptionsmover schließlich benötigt hier in beiden Fällen 90ms um dabei ein Ergebnis zu berechnen, das nur 62,8% der Kosten des MILP beträgt. Selbst bei einer mittleren Instanz mit 204 WEA, 14 Umspannstationen sowie einer Tightness von 0,85 benötigt immer noch in allen Fällen unter 2min, der Adoptionsmoveralgorithmus mit MST-Initialiserung benötigt dabei 119.7s um damit ein Ergebnis zu berechnen dessen Kosten nur 96%% der Kosten des MILP betragen. Erst bei den sehr großen Instanzen wird der Algorithmus langsam, eine Instanz mit 499 WEA, 23 Umspannstationen und einer Tightness von 0,98 benötigt für die Berechnung des Adoptionsmoveralgorithmus mit MST-Initialiserung 105.77 Minuten das Ergebnis ist dann nur 94% der Kosten des MILP. In 4.5 sind die durchschnittlichen Zeiten gegenüber der Anzahl an WEA aufgetragen. Zur Übersicht sind in 4.6 die Zeiten auch noch für die Instanzen mit $|V_T| \leq 200$ gezeigt. Dabei kann man erkennen, dass der Adoptionsmoveralgorithmus mit MST-Initialiserung am langsamsten ist, dies liegt daran, dass bei jedem betrachteten Nachbarknoten $n \in V_T$ von $t \in V_T$ alle Untermengen der Kinder, die nahe genug an t sind, betrachtet werden. Im schlimmsten Fall können das $2^{d^+_G(n)}$ verschiedene Untermengen sein.

Die Multilevel-Quality-Threshold Varianten sind dabei noch deutlich schneller und brauchen selbst für die größten Instanzen meiner Fallstudien nur 12,75 Minuten. In 4.7 ist die

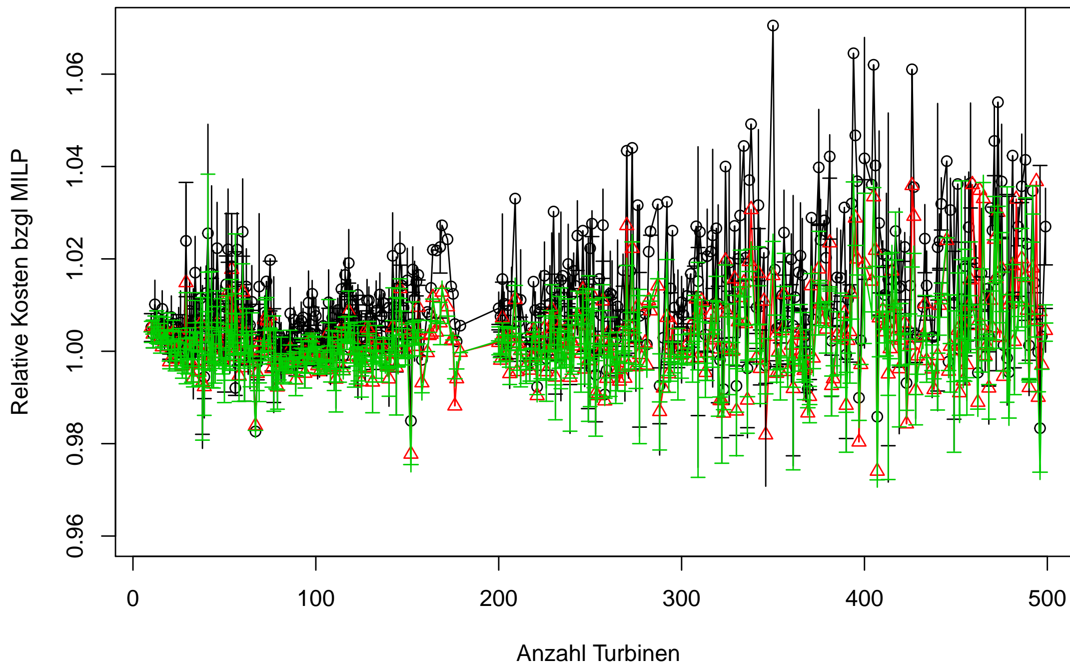

Abbildung 4.3: Ein Vergleich der drei Movervarianten bei der MST-Initialiserung. Schwarz ist der Standardmover, Rot der Single-Move-Algorithmus, Grün der Adoptionsmover. Die Kosten sind dabei immmer in Prozent der Kosten der Lösung des MILP aufgetragen, als der Mittelwert und die Fehlerbalken sind der Standardfehler

Rechenzeit dieser Algorithmen sowie des Minimalen Spannbaums über die Anzahl an Turbinen aufgetragen.

Als Beispielinstanz sind in 4.8, 4.8 und 4.10 die Ergebnisse für eine Instanz mit 143 WEA, 8 Umspannstationen und einer Tightness von 0,993 jeweils in allen Movervariaten, den beiden Initialisierungsvarianten sowie den Multilevel Quality Threshold Algorithmen und dem Minimalen Spannbaum gezeigt.

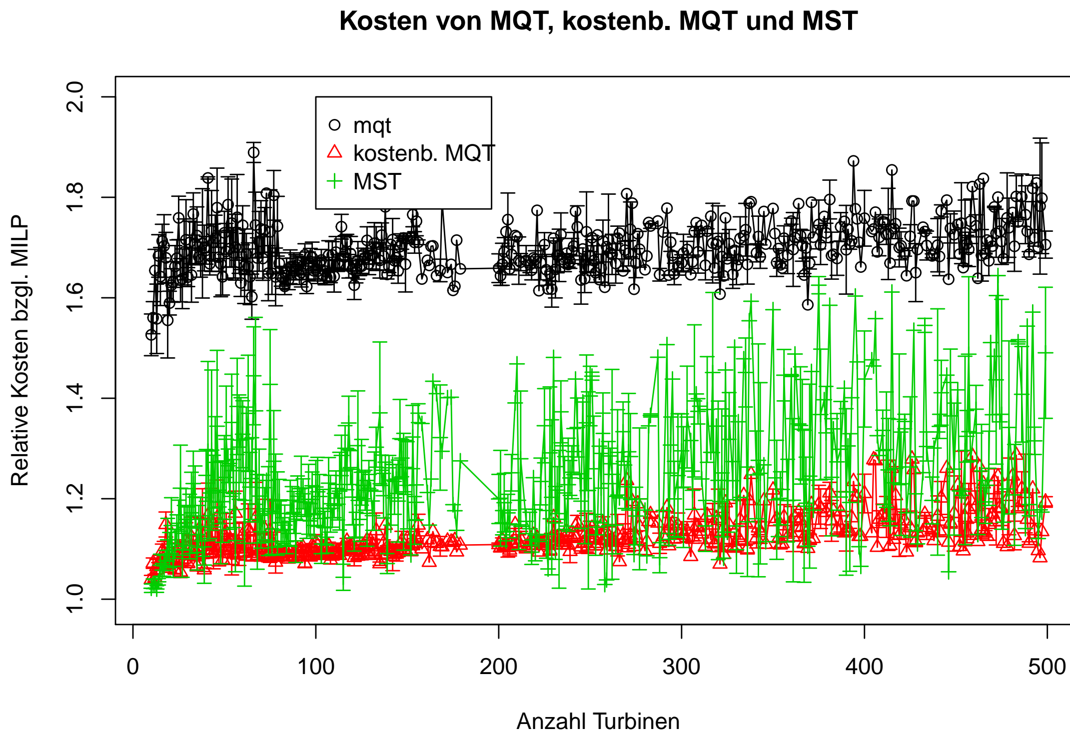

Abbildung 4.4: Ein Vergleich der beiden Multilevel-Quality-Threshold-Varianten sowie des Minimalen Spannbaums. Schwarz ist der Standard-Multilevel-Quality-Threshold-Algorithmus, Rot der kostenbasierte Multilevel-Quality-Threshold-Algorithmus, Grün der Minimale Spannbaum. Die Kosten sind dabei immmer in Prozent der Kosten der Lösung des MILP aufgetragen, als der Mittelwert und die Fehlerbalken sind der Standardfehler

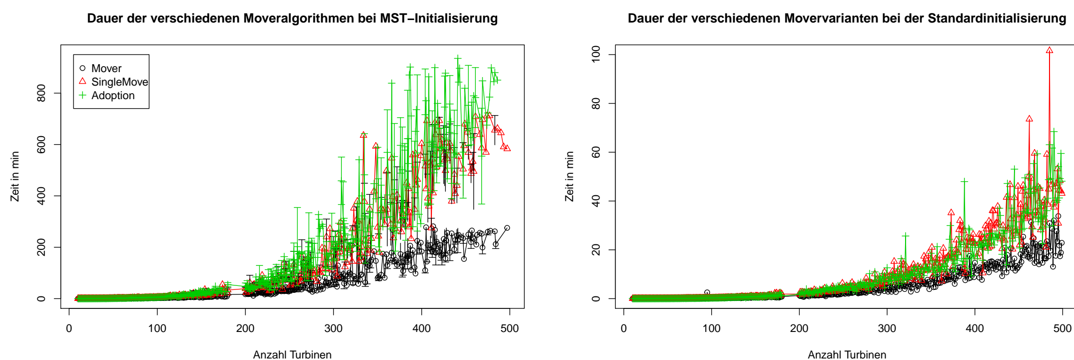

Abbildung 4.5: Zeit die die Algorithmen jeweils zum Berechnen der einzelnen Instanzen benötigt haben in Minuten aufgetragen über die Anzahl an WEA. Die Berechnungen wurden alle auf einem AMD Opteron^{TM}Processor 6172 durchgeführt. Die Fehlerbalken zeigen den Standardfehler

Algorithm 4.5: Pseudocode, der die Adoptions-Move-Operation beschreibt

```
   Input: flowOld, cost, u, oldParent, newParent, childrenToAdopt, n_S, n_C
   Output: flowNew, cost
   // Initialisierung
 1 flowNew = flowOld
 2 flowNew[u,oldParent] = 0
 3 cost = cost- c_cab(flowOld[u, oldParent]) · len(u,oldParent )
 4 flowNew[u,newParent] = 1
 5 cost = cost + c_cab(1) · len(u,newParent ) if !isForest(flowNew) then
 6     return none
   // 1 vom alten Pfad abziehen
 7 it = oldParent
 8 while isTurbine(it) do
 9     flowNew[it, parent(it)] = flowOld[it, parent(it)] - 1
10     cost = cost −(c_cab(flowOld[it, parent(it)]) + c_cab(flowNew[it,
        parent(it)])) · len(it, parent(it))
11     it = parent(it)
   // alle Kinder von u werden von oldParent adoptiert
12 forall c ∈ children(u) do
13     flowNew[c,u] = 0
14     flowNew[c,oldParent]= flowOld[c, u]
15     cost = c_cab(flowOldc, u) · (len(c,oldParent ) − len(c, u))
   // 1 auf neuen Pfad aufaddieren
16 it = newParent
17 while isTurbine(i) do
18     flowNew[it, parent(it)] = flowOld[it, parent(it)] + 1
19     if isTurbine(parent(it)) ∧ flowNew[it, parent(it)] ≥ n_S then
20         return None
21     if !isTurbine(parent(it)) ∧ flowNew[it, parent(it)] ≥ n_C then
22         return None
23     cost = cost −(c_cab(flowOld[it, parent(it)]) + c_cab(flowNew[it,
        parent(it)])) · len(it, parent(it))
   // childrenToAdopt von u adoptieren
24 forall c ∈ children(u) do
25     flowNew[c,newParent] = 0
26     flowNew[c,u]= flowOld[c, newParent]
27     cost = c_cab(flowOldc, newParent) · (len(c, u) − len(c,newParent ))
28 return flowNew, cost
```

Algorithm 4.6: Pseudocode der das Multilevel Quality Threshold Verfahren beschreibt.

```
   Input: initialMultiplyer, G = (V, E)
   Output: f
   // Initialisierung
 1 E' = ∅
 2 threshold = initialMultiplyer · MINDISTANCE(G)
 3 workingList = V_T
 4 forall v ∈ workingList do
 5     REPRESENTATIVES(v) = {v}
 6 maxClusterSize = 1
   // main loop
 7 while maxClusterSize ≤ n_C ∧ threshold < stop do
 8     cluster = QUALITYTHRESHOLDCLUSTERING(workingList, threshold)
 9     forall c ∈ cluster do
           // Repräsentanten bestimmen
10         distance = ∞
11         forall v ∈ c do
12             if len(v,CLOSESTSUBSTATION(v)) < distance then
13                 distance = len(v,CLOSESTSUBSTATION(v))
14                 rep = v
           // alle Knoten in c mit rep in Sternform verbinden
15         forall v ∈ c do
16             if v neq rep then
17                 E' = E' ∪ (v,rep )
           // alle Knoten in c außer rep aus workingList löschen
18         forall v ∈ c do
19             if v neq rep then
20                 workingList.ERASE(v)
           // Abbildung REPRESENTATIVES aktualisieren
21         forall v ∈ c do
22             clust = clust ∪ REPRESENTATIVES(v)
23             REPRESENTATIVES(v) = ∅
24         REPRESENTATIVES(rep) = clust
       // maxClusterSize bestimmen
25     maxClusterSize = 0;
26     forall c ∈ REPRESENTATIVES do
27         if |REPRESENTATIVES(c)| > maxClusterSize then
28             maxClusterSize = |REPRESENTATIVES(c)|
29     threshold = threshold ·2
30 forall t ∈ V_t do
31     if INDEG(t) = 0 then
32         E' = E' ∪ (v,CLOSESTSUBSTATION(v)))
33 f = COMPUTEFLOW(G_Kabellayout)
34 return f
```

Algorithm 4.7: Pseudocode der das Multilevel Quality Threshold Verfahren mit Kostenfunktion beschreibt.

```
   Input: initialMultiplyer, G = (V, E)
   Output: f
   // Initialisierung
1  E′ = ∅
2  threshold = initialMultiplyer · minDistance(G) · c_cab(1)
3  workingList = V_T
4  forall v ∈ workingList do
5  |   representatives(v) = {v}
6  maxClusterSize = 1
   // main loop
7  while maxClusterSize ≤ n_C ∧ threshold < stop do
8  |   cluster = qualityThresholdClusteringCost(workingList, threshold)
9  |   forall c ∈ cluster do
   |   |   // Repräsentanten bestimmen
10 |   |   distance = ∞
11 |   |   forall v ∈ c do
12 |   |   |   if len(v,closestSubstation(v)) < distance then
13 |   |   |   |   distance = len(v,closestSubstation(v))
14 |   |   |   |   rep = v
   |   |   // MST des Teilgraphen, der durch c induziert wird
15 |   |   G_c := Teilgraph, der durch die Knoten in c induziert wird.
16 |   |   G′_c = (E′_c, c) = prim(G_c)
17 |   |   E′ = E′ ∪ E′_C
   |   |   // alle Knoten in c außer rep aus workingList löschen
18 |   |   forall v ∈ c do
19 |   |   |   if v neq rep then
20 |   |   |   |   workingList.erase(v)
   |   |   // Abbildung representatives aktualisieren
21 |   |   forall v ∈ c do
22 |   |   |   clust = clust ∪ representatives(v)
23 |   |   |   representatives(v) = ∅
24 |   |   representatives(rep) = clust
   |   // maxClusterSize bestimmen
25 |   maxClusterSize = 0;
26 |   forall c ∈ representatives do
27 |   |   if |representatives(c)| > maxClusterSize then
28 |   |   |   maxClusterSize = |representatives(c)|
29 |   threshold = threshold ·2
30 forall t ∈ V_t do
31 |   if indeg(t) = 0 then
32 |   |   E′ = E′ ∪ (v,nearestValidNeighbour(v)))
33 f = computeFlow(G_Kabellayout)
34 return f
```

Benchmark Menge	$\lvert V_T\rvert$		$\lvert V_S\rvert$		β		γ		k	ϵ	$\lvert\mathcal{N}\rvert_i$
	min	max	min	max	min	max	min	max			
$\mathcal{N}_1$	10	79	1	1	0,7	1,1	1	1,88	6	1,1	500
$\mathcal{N}_2$	20	79	2	7	0,7	1	0,8	1,03	6	1,1	500
$\mathcal{N}_3$	80	180	4	9	0,5	1	0,81	1,04	6	1,1	1000
$\mathcal{N}_4$	200	499	10	39	0,4	1	0,81	1,03	6	1,1	1000
$\mathcal{N}_5$	80	180	4	9	0,5	1	0,8	1,03	$n-1$	-	1000

Tabelle 4.1: Die Benchmarkmengen für meine Fallstudien. Die Mengen $\mathcal{N}_1 - \mathcal{N}_4$ haben eingeschränkte Kantenmengen, die Menge $\mathcal{N}_5$ nicht. Die Parameter $V_T, V_S, \beta, \gamma, k, \epsilon$ sind die WEA-Menge, Umspannstationenmenge, das Seitenverhältnis, die Tightness, die k-nächsten Nachbarn und der Wert für die zusätzlichen Kanten E', wie in 4.2.1 beschrieben.

Kabeltyp	Kapazität cap$(\cdot)$	Kosten pro Einheit $c_{\text{cab}}(\cdot)$
1	5	20
2	8	25
3	12	27
4	15	41

Tabelle 4.2: Die Kabeltypen, die ich verwendet habe, basierend auf Berzan et al [BVMO11]

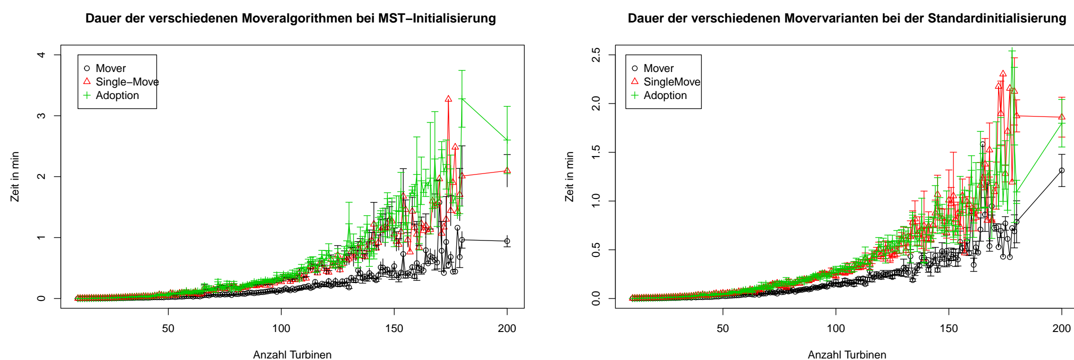

Abbildung 4.6: Zeit, die die Algorithmen jeweils zum Berechnen der einzelnen Instanzen benötigt haben in Minuten aufgetragen über die Anzahl an WEA, beschränkt auf kleine Instanzen mit $|V_T| \leq 200$. Die Berechnungen wurden alle auf einem AMD Opteron$^{\text{TM}}$Processor 6172 durchgeführt. Die Fehlerbalken zeigen den Standardfehler

Initialiserung:	Mover		SingleMove		Adoptionsmover		MQT	kostenb. MQT	MST
	Std.	MST	Std.	MST	Std.	MST			
Durchschnitt	1,067	1,020	1,062	1,008	1,058	1,007	1,698	1,124	1,243
Standardfehler	0,002	0,001	0,002	0,001	0,002	0,001	0,002	0,002	0,004
Median	1,038	1,007	1,034	1,001	1,031	1,001	1,683	1,103	1,188
unteres Quartil	1,019	1,000	1,017	0,995	1,009	0,995	1,636	1,082	1,088
oberes Quartile	1,067	1,019	1,062	1,008	1,062	1,007	1,742	1,131	1,338
Varianz	0,008	0,003	0,007	0,001	0,007	0,001	0,012	0,007	0,045
Standardabw.	0,088	0,059	0,084	0,039	0,086	0,039	0,107	0,084	0,212
Minimum	0,977	0,966	0,973	0,965	0,965	0,965	1,140	1,001	0,990
Maximum	1,618	1,745	1,666	1,496	1,640	1,590	2,277	1,976	3,222

Tabelle 4.3: Statistische Daten über meine Fallstudien im Vergleich mit den Kosten, die durch das MILP berechnet werden.

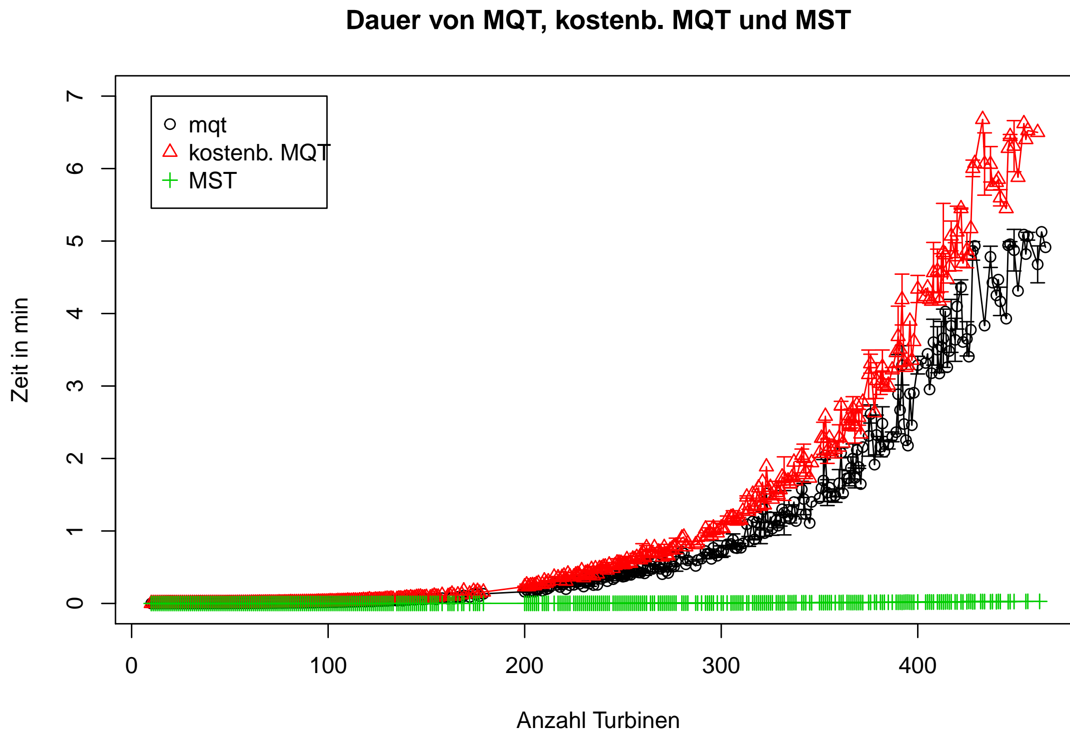

Abbildung 4.7: Zeit, die die Multilevel-Quality-Threshold Varianten und des Minimalen Spannbaums jeweils zum Berechnen der einzelnen Instanzen benötigt haben in Minuten aufgetragen über die Anzahl an WEA. Die Berechnungen wurden alle auf einem AMD Opteron™Processor 6172 durchgeführt. Die Fehlerbalken zeigen den Standardfehler

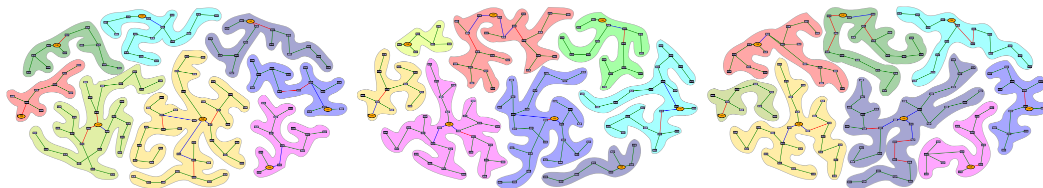

Abbildung 4.8: Ergebnisse der Moveralgorithmen in der Standardinitialiserung, links der Standardmover mit Kosten von 265094, in der Mitte der Single-Move-Algorithmus mit Kosten von 263858, rechts der Adoptionsmover mit Kosten von 264532

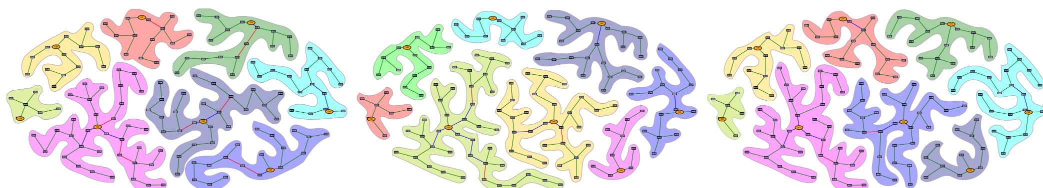

Abbildung 4.9: Ergebnisse der Moveralgorithmen bei der MST-Initialiserung, links der Standardmover mit Kosten von 252558, in der Mitte der Single-Move-Algorithmus mkt Kosten von 250121, rechts der Adoptionsmover mit Kosten von 250540

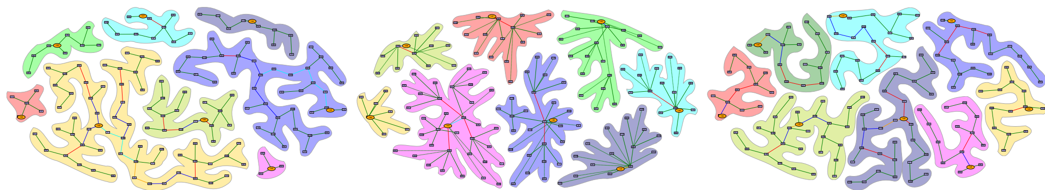

Abbildung 4.10: Ergebnisse der Multilevel-Quality-Threshold Varianten und des Minimalen Spannbaums, links der Standard-Multilevel-Quality-Threshold-Algorithmus mit Kosten von 431081, in der Mitte der kostenbasierte Standard-Multilevel-Quality-Threshold-Algorithmus mit Kosten von 276322, rechts der Minimale Spannnbaummit Kosten von 321650

5. Zusammenfassung

In meiner Arbeit habe ich das Problem des Kabellayouts in einer Windfarm betrachtet. Dies ist das Problem wie die unterschiedlichen Komponenten einer Windfarm (WEA, Umspannstationen, Umspannwerke) miteinander verbunden werden, also welche Komponenten mit einem Kabel verbunden sind, und welches Kabel aus einer gegebenen Menge an Kabeln verwendet wird. Dabei habe ich mehrere Varianten eines Moveralgorithmus sowie das von Dutta et. al. [DO11] beschriebene Multilevel Quality Threshold Clustering und eine Variante davon betrachtet.
Dabei wurde die Windfarm immer als Graph $G = (V, E)$ modelliert, wobei die Knotenmenge sich aufteilen lässt in die Menge der WEA V_T und die Menge der Umspannstationen V_S, Die Kantemenge E stellt die Menge der möglichen Verbindungen dar. Der Strom, der auf diesen Verbindungen fließt, wurde als Fluss $f(e)$ auf den Kanten in E modelliert. Auf Basis dieses Flusses auf der Kante $e \in E$ lässt sich dann das verwendete Kabel auf dieser Kante auswählen mit der einer Funktion $\kappa : E \to K$, dabei ist K die Menge an Kabeltypen. Durch diesen Fluss f und die Kabelzuordnungsfunktion κ lassen sich dann die Kosten der Windfarm berechnen durch die Kostenfunktion

$$c(\kappa, f) = \sum_{e \in E} c_{cab}(\kappa(e)) \cdot len(e)$$

Der Moveralgorithmus ist ein Algorithmus, der lokale Verbesserungen sucht in dem für jede WEA $t \in V_T$ betrachtet wird, ob sich die Kostenfunktion verbessert, wenn die Senke des Flusses der von t ausgeht, zu einem anderen Knoten $v \in V$ umgeleitet wird. Dabei habe ich drei Varianten davon untersucht.
Der **Standardmover** bei dem der Fluss von der WEA $t \in V_T$, der bisher zum Knoten $p \in V$ fließt, umgeleitet wird zum Knoten $v \in V$. Dabei werden alle Kinder von t beibehalten. Dies wird für jede WEA $t \in V_T$ mit allen Nachbarn $n \in N(t)$ durchgeführt und immer die beste Vebesserung für t gespeichert und für die Suche des nächsten Knoten verwendet.
Der **Single-Mover** bei dem eine zweistufige Suche verwendet wird. Zunächst wird mit dem Standardmover gesucht bis dieser keine Verbesserungen mehr findet. Danach wird für alle WEA $t \in V_T$ überprüft, ob sich die Kostenfunktion verbessert, wenn nur t bewegt wird. Dies bedeutet, dass der Fluss der Kinder von t umgeleitet wird zum ursprünglichen Elter von$t$ und für t ein neuer Elter gesucht wird ohne dass die Kinder mit bewegt werden. Falls dies eine Verbesserung findet, wird danach wieder mit dem Standardmover gesucht.
Die dritte Variate ist der **Adoptionsmover**. Bei diesem wird wieder zweistufig gearbeitet und zunächst mit dem Standardmover gesucht. Falls dieser keine Verbesserungen mehr findet, wird als Move-Operation adoptiert. Es wird also sowohl der Fluss der Kinder von t

wieder zu dem Elter von t umgeleitet als auch der Fluss einer Untermenge der Kinder von v, also dem neuen Elter von t, zu t umgeleitet.
Außerdem habe ich für alle diese Varianten verschiedene Initiallösungen getestest. Zunächst die **Standardinitialisierung** bei der die WEA zufällig gleichmäßig zu Umspannstationen zugeordnet werden.
Schließlich noch die **MST-Initialisierung** bei der ein Minimaler Spannbaum mit Prims Algorithmus berechnet wird.
Weiterhin wurde der Multilevel Quality Threshold Clustering Algorithmus aus [DO11] implementiert sowie eine Verbesserug davon, die die Kosten der Kabel und damit die Größe der schon berechneten Cluster berücksichtigt, außerdem wird in der verbesserten Variante die Verkabelung innerhalb der Cluster als Minimaler Spannbaum berechnet.
Dabei zeigte sich, dass beide Verbesserungen des Standardmoveralgorithmus ähnlich gut sind und jeweils in über 44% der Instanzen meiner Fallstudien besser waren als das MILP. Generell sind alle meine Varianten konkurrenzfähig zum MILP, und dabei deutlich schneller. Bei allen Instanzen unter 300 WEA wurde das Ergebnis in unter 15 Minuten berechnet. Die Multilevel-Quality-Threshold-Varianten sind weniger gut als die Moveralgorithmen, die Kosten der kostenbasierte Variante sind im Median allerdings immer noch nur 110,3% der Kosten des MILP. Dabei sind die Multilevel-Quality-Threshold-Varianten noch schneller als die Moveralgorithmen und selbst die größten Instanzen sind in unter 13 Minuten berechnet.

5.1 Zukünftige Arbeit

Da die Ergebnisse von der Initiallösung abhängen, lassen sich mit einer besseren Initialisierungslösung vermutlich noch bessere Ergebnisse finden. Eine Idee hierfür wäre zm Beispiel statt einem minmalen Spannbaum als Initiallösung einen beschränkten minimalen Spannbaum (CMST) zu berechnen. Eine andere Idee wäre es die in der kostenbasierten Variante berechneten Graphen als Initialisierungslösung zu verwenden.
Außerdem sind auch noch Verbesserungen des Adoptionsmover möglich. Der Algorithmus könnte vermutlich noch deutlich beschleunigt werden durch eine Parallelisierung.
Wenn man die Kostenschätzung bei der kostenbasierten Variante des Multilevel-Quality-Threshold-Algorithmus verbessert, könnte dies ebenfalls noch deutlich bessere Ergebnisse geben.

Literaturverzeichnis

[BHSW15] Ulrik Brandes, Michael Hamann, Ben Strasser und Dorothea Wagner: *Fast Quasi-Threshold Editing.* In: *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA'15)*, Lecture Notes in Computer Science, Seiten 251–262. Springer, 2015.

[BVMO11] Constantin Berzan, Kalyan Veeramachaneni, James McDermott und Una May O'Reilly: *Algorithms for Cable Network Design on Large-scale Wind Farms.* MSRP Technical Report, 2011.

[CGJ+12] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Karsten Klein und Petra Mutzel: *The Open Graph Drawing Framework (OGDF).* CRC Press, 2012. to appear.

[DO11] S Dutta und TJ Overbye: *A clustering based wind farm collector system cable layout design.* In: *Power and Energy Conference at Illinois (PECI), 2011 IEEE*, Seiten 1–6. IEEE, 2011.

[Kal13] Martin Kaltschmitt: *Erneuerbare Energien : Systemtechnik, Wirtschaftlichkeit, Umweltaspekte*, 2013, ISBN 978-3-642-03249-3.

[Kru12] Sven O. Krumke: *Graphentheoretische Konzepte und Algorithmen*, 2012, ISBN 978-3-8348-2264-2.

[LRWW17] Sebastian Lehmann, Ignaz Rutter, Dorothea Wagner und 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, Seiten 203–215, New York, NY, USA, 2017. ACM, ISBN 978-1-4503-5036-5. http://doi.acm.org/10.1145/3077839.3077843.

[Mac11] David J. C. MacKay: *Information theory, inference, and learning algorithms.* Cambridge University Press, Cambridge [u.a.], 10. print. Auflage, 2011, ISBN 978-0-521-64298-9.

[Nat15] United Nations: *Paris Agreement*, 2015.

[NG04] M. E. J. Newman und M. Girvan: *Finding and evaluating community structure in networks.* Phys. Rev. E, 69:026113, Feb 2004. https://link.aps.org/doi/10.1103/PhysRevE.69.026113.

[Ste57] Hugo Steinhaus: *Sur la division des corps matériels en parties.* Bull. Acad. Pol. Sci., Cl. III, 4:801–804, 1957, ISSN 0001-4095.

[str17] *Zahlen und Fakten , Wirtschaftsbereiche , Energie , Erzeugung*, 2017. www.destatis.de/DE/ZahlenFakten/Wirtschaftsbereiche/Energie/Erzeugung/Tabellen/Bruttostromerzeugung.html, Accesses: 2017-06-11.

[Wag98] Prof. Dr. Wagner: *Grundlagen: Begriffe zu Graphen*, 1998.