# Aufgabe 1: Müllabfuhr

Teilnahme-ID: 60963

Tobias Hettler

24. April 2022

## Inhaltsverzeichnis

# 1 Komplexität

**Theorem 1.1.** Das Müllabfuhr Problem ist NP-Schwer.

*Beweis.* Um dies zu beweisen, reduziere ich vom multiway number partitioning Problem [1] , welches NP-Schwer ist, auf das Müllabfuhr Problem. Beim multiway number partitioning Problem sind eine Multimenge $M$ an Zahlen und eine Zahl $k$ gegeben. Das Ziel ist es, die Zahlen in $M$ in $k$ Teilmengen aufzuteilen, sodass die größte Summe aller Zahlen in einer Teilmenge minimiert wird.
Um nun eine Instanz des Müllabfuhr Problems zu erstellen, wird der Stadtplan folgendermaßen gebaut: Für jede Zahl $m \in M$ fügt man eine Straße mit der Länge $m$ hinzu, die von der Zentrale zu einer Sackgasse führt. Als Ergebnis hat man also einen Stadtplan, bei dem es nur Straßen gibt, die von der Zentrale zu einer Sackgasse führen. Wenn man jetzt das Müllabfuhr Problem löst, indem man einen Fahrplan für $k$ - in unserem Fall 5 - Tage erstellt, ist dies auch eine Lösung für das multiway number partioning Problem. Die Straßen wurden so auf die Tage verteilt, dass die längste Tagestrecke möglichst kurz ist. Mit anderen Worten: die Menge mit der größten Summe an Zahlen ist möglichst gering, also ist diese Verteilung auch eine Lösung für das multiway number partioning Problem.
Genau genommen müsste man die Summen der Strecken jeweils noch durch 2 teilen, da beim Müllabfuhr Problem jede Straße zweimal abgefahren werden müsste, da man wieder zur Zentrale zurückkommen muss.

Die Generierung einer Müllabfuhr Problem Instanz ist in polynomieller Zeit möglich. Jede Zahl in $M$ wird nur einmal betrachtet und die Straßen lassen sich in konstanter Zeit hinzufügen. Somit ist die Laufzeit der Umwandlung $\Theta(|M|)$. ■

Da das Problem NP-Schwer ist, habe ich zwei Heuristiken entwickelt und keinen optimalen Algorithmus. Der erste Algorithmus ist etwas langsamer liefert dafür aber bessere Ergebnisse.

# 2 Algorithmus 1

## 2.1 Lösungsidee

Der Algorithmus läuft in drei Schritten ab.

1. Generieren einer Ausgangslösung
2. Verbesserung dieser Lösung durch einen Hill Climbing Algorithmus
3. Versuch das mehrfache Abfahren von Straßen zu minimieren

### 2.1.1 Repräsentation

Das Straßennetz repräsentiere ich als einen Graphen. Daher werde ich im Folgenden von Knoten statt Kreuzungen/Sackgassen und Kanten statt Straßen sprechen.

**Definition 2.1** (Lösung)**.** Eine Lösung ordnet jede Kante einem Tag zu. Zudem ist die Reihenfolge der Kanten an einem Tag durch die Lösung festgelegt.

[1] `https://en.wikipedia.org/wiki/Multiway_number_partitioning`

**Definition 2.2** (Wert einer Lösung)**.** Der Wert einer Lösung ist die Länge der längsten Tagesstrecke. Folglich ist eine Lösung mit niedrigerem Wert besser als eine mit höherem Wert.

**Definition 2.3** (Distanz zwischen zwei Knoten)**.** Die kürzeste Distanz zwischen zwei Knoten $v$ und $u$ im Graphen gebe ich im Folgenden mit $d(v, u)$ an.

### 2.1.2 Generierung der Ausgangslösung

Der Algorithmus basiert auf der Idee, dass Straßen, die nah beieinander liegen auch möglichst am gleichen Tag abgefahren werden sollten. Zudem sollten an jedem Tag ungefähr gleich viele Straßen abgefahren werden. Damit wird zwar vernachlässigt, dass Straßen unterschiedlich lang sind, jedoch ist das eine notwendige Vereinfachung. Wenn man versucht die Gesamtstrecke der verschiedenen Tage ungefähr auf den gleich Wert zu bekommen, handelt es sich wiederum um das oben beschriebene multiway number partitioning Problem, welches NP-Schwer ist.

Jedem Tag werden also ungefähr $\frac{m}{5}$ von insgesamt $m$ Straßen zugeordnet. Genauer gesagt werden den ersten 4 Tagen $\lfloor \frac{m}{5} \rfloor$ Straßen zugeordnet und dem fünften Tag der Rest.

Der Algorithmus ist im Grunde eine Tiefensuche. Trifft man mit dieser auf einen Knoten, werden alle Kanten, die mit diesem Knoten verbunden sind dem Tag $i$ zugeordnet. Wenn dem $i$-ten Tag $\frac{m}{5}$ Kanten zugeordnet wurden, geht man zum $i + 1$-ten Tag.

**Algorithmus 1 :** Generierung der anfänglichen Lösung

```
Stack s // Stack nach FIFO Prinzip
while s nicht leer do
    v ← s.top() // Nehme obersten Knoten vom Stack
    forall Nachbarn u von v do
        if Kante v - u noch nicht zugewiesen then
            // Wenn Tag i bereits ⌊m/5⌋ Straßen zugeordnet sind, gehe zu Tag i + 1
            Weise Kante v - u Tag i zu
        end if
        if Knoten u unbekannt then
            // Füge Knoten u dem Stack hinzu
            s.push(u)
            markiere u als bekannt
        end if
    end forall
end while
```

### 2.1.3 Hill Climbing

Nachdem im ersten Schritt eine Ausgangslösung generiert wurde, wird diese nun mittels eines Hill Climbing Algorithmus weiter verbessert. Dies läuft folgendermaßen ab: In jeder Iteration wird die aktuelle Lösung leicht verändert. Es wird also ein Nachbarknoten im Graphen des Suchraums genommen. Verbessert dies die Lösung, wird die Veränderung behalten, ansonsten verworfen. Um die Anzahl der Iterationen zu bestimmen verwende ich folgende Formel:

$$I = 40000 \cdot e^{-(n+m)\cdot\alpha}$$

$n$ ist hier die Anzahl der Knoten und $m$ die Anzahl der Kanten im Graphen. Für den Parameter $\alpha$ habe ich $4,6 \cdot 10^{-4}$ gewählt. Dies sorgt dafür, dass das Programm beim größten Beispiel 8 nach ca. 5150 Iterationen noch in unter einer Sekunde terminiert und bei den kleineren Beispielen ungefähr 40000 Iterationen ermöglicht.

**Berechnung der Nachbarn** Der Wert einer Lösung hängt nur von der längsten Tagesstrecke ab. Daher macht es Sinn, beim Berechnen eines Nachbarn etwas an der längsten Tagestrecke zu verändern.

**Algorithmus 2 :** Hill Climbing Algorithmus

```
Loesung s // Ausgangslösung, wie oben beschrieben generiert
for i ← 1, iterationen do
    s' ← nachbar(s)
    if s' besser als s then
        s ← s'
    end if
end for
```

Daher wird, um einen Nachbarn von einer Lösung zu berechnen, als erstes eine zufällige Kante aus der längsten Tagesstrecke ausgewählt und einer anderen, zufällig ausgewählten Tagesstrecke zugeordnet.
Als zweiter Schritt werden zufällig zwei Kanten innerhalb der bisherigen längsten Tagesstrecke getauscht. Dadurch verändert sich die Reihenfolge in der die Kanten abgefahren werden.

**Berechnung des Wertes einer Lösung** Um zwei Lösungen vergleichen zu können, muss der Wert für beide berechnet werden.

**Lemma 2.1.** Der kürzeste Weg im Graphen von einem Knoten $v$, der die Kante $(u, x)$ passiert, hat die Länge:

$$L(v,(u,x)) = \begin{cases} d(v,u) + |(u,x)| & \text{für} \quad d(v,u) \leq d(v,x) \\ d(v,x) + |(x,u)| & \text{für} \quad d(v,u) > d(v,x) \end{cases}$$

Die Notation $|(u, x)|$ gibt hier die Länge der Kante $|(u, x)|$ an.

*Beweis.* Jeder Pfad, der die Kante $(u, x)$ passiert, muss entweder von $u$ nach $x$ gehen oder von $x$ nach $u$. In einem ungerichteten Graphen gilt $|(u, x)| = |(x, u)|$. Wichtig ist, dass man hier $|(u, x)|$ und nicht $d(u, x)$ nimmt. Denn $d(u, x)$ ist möglicherweise kürzer als der Weg über die Kante $(u, x)$. Somit ist die Distanz nur durch $d(v, u)$ oder $d(v, x)$ unterschieden. Wenn also $d(v, u) \leq d(v, x)$ muss auch gelten $d(v, u) + |(u, x)| \leq d(v, x) + |(x, u)|$. Wenn man davon ausgeht, dass der Graph keine negativen Kantengewichte hat, würde jede Kante, die man nach der Kante $(u, x)$ anhängt, die Länge nur vergrößern. ■

**Folgesatz 2.1.1.** Der kürzeste Weg von einem Knoten $v$, der die Kante $(u, x)$ passiert, endet bei Knoten

$$T(v,(u,x)) = \begin{cases} x & \text{für} \quad d(v,u) \leq d(v,x) \\ u & \text{für} \quad d(v,u) > d(v,x) \end{cases}$$

Nun kann man eine Formel für die Länge $S$ einer Tagesstrecke aufstellen. Zunächst braucht man jedoch eine andere Formel für $T$. Es sei $M$ die Liste der Kanten der Tagestrecke. Die Kanten in $M$ hängen nicht unbedingt direkt anneinander, darum müssen auch die Strecken zwischen diesen mit einberechnet werden. Zunächst lässt sich $T$ als eine Rekursion darstellen. $T_n = T(T_{n-1}, m_n)$ mit $T_0 = 0$, da 0 die Zentrale ist. $T_n$ ist der Knoten an dem man sich befindet, nachdem die $n$-te Kante aus $M$ durchfahren wurde. Hierbei ist $m_n$ die $n$-te Kante aus $M$. Somit gilt die Formel für $1 \leq n \leq |M|$.

Damit lässt sich jetzt eine Formel für die Länge $S$ einer Tagesstrecke finden. Es sei $S_n$ die Länge der Strecke zwischen der $n-1-$ten und der $n$-ten Kante. $S_n = L(T_{n-1}, m_n)$. Mit $S_n$ definiert lässt sich jetzt die Gesamtlänge einer Tagestrecke berechnen:

$$S = (\sum_{i=1}^{n} L(T_{i-1}, m_i)) + d(T_n, 0)$$

Es muss noch $d(T_n, 0)$ addiert werden, da jede Tagesstrecke wieder in der Zentrale enden muss.

Mit der Formel für $S$ kann man jetzt den Wert $W$ einer Lösung $l$ berechnen. $l_i$ ist hier die $i$-te Tagesstrecke der Lösung.

$$W = \min_{\forall l_i \in l}(S(l_i))$$

#### 2.1.4 Versuch das mehrfache Abfahren von Straßen zu minimieren

Wenn während einer Tagesstrecke $l_i$ nacheinander zwei Kanten abgefahren werden sollen, die nicht einen gemeinsamen Knoten haben, muss man zwangsläufig andere Kanten dazwischen abfahren, um von der einen Kante zur anderen zu kommen. Diese Kanten, die zwischendrin abgefahren werden, sind jedoch möglicherweise einer anderen Tagesstrecke $l_j$ zugeordnet. Das heißt, wenn diese Kanten sowieso während der Tagesstrecke $l_i$ abgefahren werden, müssen sie nicht noch einmal während Tag $j$ abgefahren werden. Folglich können sie aus der Tagesstrecke $l_j$ gelöscht werden.

Dazu werden zunächst die Tagesstrecken nach aufsteigender Länge sortiert. Für die $i$-te Tagesstrecke werden alle Kanten gespeichert, die „zwischendrin" abgefahren werden, aber nicht der $i$-ten Tagesstrecke zugeordnet sind. Dann werden bei allen Tagestrecken mit Index $j > i$ die Kanten gelöscht, die bei $i$ „zwischendrin" abgefahren werden.

Dadurch, dass die Tagesstrecken sortiert wurden, werden die Chancen, das die längste Tagesstrecke verkürzt wird, maximiert.

### 2.2 Umsetzung

Das Programm habe ich in C++ implementiert.

#### 2.2.1 Repräsentation einer Lösung

Eine Lösung wird als ein 2 dimensionaler *std::vector* aus Integern repräsentiert. Der äußere Vector besteht aus 5 Vectoren, einen für jeden Tag. Jeder Vector für einen Tag enthält die Indizes der Kanten, die an diesem Tag abgefahren werden sollen. Um den Code besser lesbar zu machen gibt es die Zeile *typedef std::vector<std::vector<int> > loesung*, die einem zweidimensionalen Integer Vector den Alias „loesung" zuweist.

Wenn im Code also irgendwo ein Lösungsvector gebraucht wird, verwende ich diesen Alias.

#### 2.2.2 Berechnung von d(u, v)

Um die Distanz $d(v, u)$ zwischen beliebigen Knoten effizient bestimmen zu können, wird diese vorberechnet. Hierzu wird von jedem Knoten einmal ein Dijkstra Algorithmus ausgeführt. Wird der Algorithmus also z.B. von Knoten $i$ ausgeführt, lässt sich dadurch $d(i, x)$ für alle Knoten $x$ berechnen. Das Ergebnis von $d(v, u)$ wird im *std::vector* distanz[v][u] gespeichert. Die Funktion $d(v, u)$ lässt sich danach mittels des Distanz Vectors in $\mathcal{O}(1)$ berechnen.

Zusätzlich wird noch ein *std::vector* parent[v][u] gefüllt, wobei parent[v][u] den nächsten Knoten auf dem kürzesten Pfad zwischen $v$ und $u$ angibt.

#### 2.2.3 Wert einer Lösung berechnen

Um die Funktion $L(v, (u, x))$ zu berechnen, gibt es die Methode *berechneStrecke()*. Diese gibt einen *std::tuple* der Form $\{L(v, (u, x)); T(v, (u, x))\}$ zurück. So kann man die Methode für die erste Kante einer Tagesstrecke aufrufen und erhält die Länge der kürzesten Strecke von einem gegebenen Knoten zu dieser Kante und den Knoten bei dem man sich danach befindet. Mit diesem Knoten lässt sich die Methode dann wieder für die nächste Kante aufrufen.

Dieses wiederholte Aufrufen der *berechneStrecke()* Methode um die Gesamtlänge $S$ einer Tagesstrecke zu berechnen, wird in der Methode *berechneTagesstrecke()* umgesetzt. Diese gibt die Gesamtlänge $S$ als *long long* zurück.

Um den Wert einer Lösung zu berechnen gibt es die Methode *berechneWert()*, die die *berechneTagesstrecke()* Methode für jeden Tag aufruft. Sie gibt die Länge der längsten Tagesstrecke zurück.

#### 2.2.4 Umwandeln der Lösung

Nachdem der oben beschriebene Algorithmus ausgeführt wurde, hat man noch keine richtige Lösung. Man hat lediglich eine Zuordnung von Kanten zu Tagen in einer bestimmten Reihenfolge. Diese Zuordnung

muss nun noch in eine richtige Lösung umgewandelt werden, also in einen Pfad für jeden Tag, der abgefahren werden kann. Mittels des zuvor berechneten *parent* Vectors kann der kürzeste Pfad zwischen zwei beliebigen Knoten rekonstruiert werden. Für jede Tagesstrecke mit der Menge $M$ an Kanten wird also zunächst der Pfad von der Zentrale 0 zu $T_1 = T(0, m_1)$ berechnet. Danach der Weg von $T_1$ zu $T(T_1, m_2)$ usw. Am Ende muss man noch den Pfad von $T_n$ zur Zentrale 0 berechnen, da jede Tagesstrecke wieder bei der Zentrale enden muss. Diese Pfade können dann zu einem zusammengefügt werden, was die Strecke für den entsprechenden Tag ist.
Diese Konvertierung ist in der Methode *konvertiereZuNormal()* implementiert.

# 3 Algorithmus 2

## 3.1 Lösungsidee

### 3.1.1 Repräsentation

Auch hier wird der Straßenplan wieder als Graph dargestellt. Jedoch wird der Graph für den zweiten Algorithmus etwas verändert. Für jede Kante wird ein zusätzlicher Knoten eingefügt. Aus einer Kante $(v, u)$ werden also zwei Kanten $(v, x)$ und $(x, u)$ wobei der Knoten $x$ neu hinzugefügt wird. Beide Kanten $(v, x)$ und $(x, u)$ haben die gleiche Länge wie die ursprüngliche Kante $(v, u)$. Und der Knoten $x$ ist nur mit $v$ und $u$ verbunden. Dies erlaubt es einfacher die Pfade zu einer der originalen Kanten zu berechnen.

### 3.1.2 Berechnung der Pfade

Zuerst wird von der Zentrale aus der kürzeste Pfad zu allen hinzugefügten Knoten berechnet, also zu den Kanten des ursprünglichen Graphen. Die Länge dieser Pfade wird so berechnet, dass sie der eigentlichen Länge entspricht, ohne die hinzugefügten Kanten. Dazu werden beim Berechnen der Distanz die Hälfte der Kanten übersprungen. Diese Pfade werden nun noch um eine Kante verlängert, sodass sie an einem der ursprünglichem Knoten enden.

Hier fällt auf, dass manche Kanten zwangsläufig abgegangen werden müssen, um zu weiter entfernten zu gelangen. Um dies auszunutzen, werden die hinzugefügten Knoten nach Entfernung zur Zentrale absteigend sortiert. In dieser sortierten Reihenfolge werden jetzt die Pfade zu den hinzugefügten Knoten, die den ursprünglichen Kanten entsprechen, berechnet. Zuerst wird also der Pfad zum am weitesten entfernten hinzugefügten Knoten berechnet. In diesem Pfad sind wahrscheinlich einige der ursprünglichen Kanten schon enthalten, zu denen also kein extra Pfad mehr berechnet werden muss. Der Pfad zu einem hinzugefügtem Knoten wird nur berechnet und gespeichert, wenn dieser nicht schon in einem der bisher berechneten Pfade enthalten ist.

Einige dieser Pfade werden nun höchstwahrscheinlich denselben Endknoten haben. Daher werden nun Paare dieser Pfade mit gleichem Endknoten verbunden, sodass sie bei der Zentrale anfangen und enden.

Danach werden immer die beiden kürzesten Pfade verbunden, bis es nur noch 5 Pfade gibt. Hier werden zwei Fälle unterschieden. Haben die beiden Pfade den gleichen Endknoten, werden sie einfach verbunden. Haben sie nicht den gleichen Endknoten, wird an die Pfade, die nicht in der Zentrale enden der Weg vom Endknoten zur Zentrale rückwärts angefügt. Danach haben beide Pfade die Zentrale als Endknoten und können verbunden werden.

Nach dieser Operation hat man 5 Pfade, einen für jeden Tag.

## 3.2 Durchführung

### 3.2.1 Repräsentation

Der Graph wird als eine Adjazenzliste in einem *std::vector* gespeichert. Um einen Pfad zu speichern, wird ein *std::pair* mit einem *std::vector* und einem *long long* verwendet. Der Vector enthält eine Liste mit Knoten - der Pfad - und der *long long* gibt die Gesamtlänge des Pfades an.

#### 3.2.2 Berechnung der Pfade

Um die kürzesten Pfade zu den Knoten zu berechnen, verwende ich auch hier wieder den Dijkstra Algorithmus. Dieser wird einmal von der Zentrale ausgeführt. Die Entfernung eines Knotens zur Zentrale wird im Array *dist* gespeichert. Zusätzlich zum *dist* Array wird auch noch ein *parent* Array gefüllt. *parent[v]* gibt den nächsten Knoten auf dem kürzesten Weg von $v$ zur Zentrale an. Hiermit lassen sich die Pfade von der Zentrale zu beliebiegen Knoten wieder rekonstruieren. Die Implementation davon befindet sich in der Methode *berechnePfade()*.

#### 3.2.3 Verbinden der Pfade

Das Verbinden der Pfade wird in der Methode *verbindePfade()* realisiert. Um Pfade mit gleichem Endknoten zu verbinden, werden die Indizes der Pfade zunächst in einer *std::unordered_map* gespeichert. Diese verwendet als Keys die Endknoten der Pfade. Als Wert enthält sie eine Liste mit Indizes aller Pfade, die den entsprechenden Endknoten haben. So kann man effizient auf alle Pfade mit einem bestimmten Endknoten zugreifen, um sie zu verbinden.

Danach werden die verbleibenden und verbundenen Pfade in eine *std::priority_queue* gespeichert. Diese hält die Pfade basierend auf ihrer Länge aufsteigend sortiert. So können effizient immer die beiden kürzesten Pfade aus der Prioritätswarteschlange genommen werden, diese verbunden werden und das Ergebnis wieder in die Prioritätswarteschlange eingefügt werden.

# 4 Komplexitätsanalyse

Im folgenden werden ich die Anzahl der Knoten als $n$ und die Anzahl der Kanten als $m$ bezeichnen. Wenn ich im folgenden vom Logarithmus spreche, meine ich immer den Logarithmus zur Basis 2.

## 4.1 Algorithmus 1

#### 4.1.1 Laufzeitanalyse

Um die anfägnliche Lösung zu generieren, muss man zunächst über alle Kanten iterieren, was einer Laufzeit von $\mathcal{O}(m)$ entspricht. Danach wird die modifizierte Breitensuche ausgeführt. In der Queue der Breitensuche befindet sich jeder Knoten genau einmal. Für jeden dieser Knoten wird über alle verbundenen Kanten iteriert. Also ist die Laufzeit die Summe aller Grade aller Knoten. Da jede Kante den Grad für genau zwei Knoten um eins erhöht, muss die Summe aller Grade aller Knoten genau $2m$ sein. Somit ist auch hier die Laufzeit $\mathcal{O}(m)$.

Um das *distanz* Array zu berechnen, wird der Dijkstra Algorithmus $n$ mal ausgeführt. Während einer Ausführung des Dijkstra Algorithmus wird maximal $n + m$ mal ein Knoten der Prioritätswarteschlange hinzugefügt und genauso oft wieder entfernt. Die Prioritätswarteschlange hat logarithmische Laufzeit, sodass sich die Laufzeit $\mathcal{O}((n + m) \log(n + m))$ mit $n + m \in \mathcal{O}(n^2)$ ergibt sich $\mathcal{O}((n + m) \log n))$. Da der Algorithmus insgesamt $n$ mal aufgerufen wird ist die Gesamtlaufzeit $\mathcal{O}(n^2 \log n)$.

Beim Hill Climbing Algorithmus wird während jeder Iteration einmal ein Nachbar berechnet und dessen Wert. Zunächst die Wert Berechnung. Die Methode *berechneStrecke* die die Funktion $L(v, (u, x))$ berechnet läuft in $\mathcal{O}(1)$, da die Distanz zwischen zwei beliebigen Knoten im Distanz Array vorberechnet wurde. Um den Wert einer Lösung zu berechnen, muss die Distanz zwischen $\mathcal{O}(m)$ Knoten berechnet werden. Also ist die Laufzeit zur Berechnung des Wertes einer Lösung $\mathcal{O}(m) \cdot \mathcal{O}(1) = \mathcal{O}(m)$.
Um einen Nachbarn einer Lösung zu berechnen, muss zunächst die längste Tagesstrecke bestimmt werden, was äquivalent zur Berechnung des Wertes der Lösung ist, also in $\mathcal{O}(m)$ läuft. Die restlichen Operationen hier sind alle schneller als $\mathcal{O}(m)$.
Folglich hat also eine Iteration des Hill Climbing Algorithmus die Laufzeit $\mathcal{O}(m)$. Bei $I$ Iterationen ist die Laufzeit des Hill Climbing Algorithmus also $\mathcal{O}(I \cdot m)$.

Am Ende muss die generierte Lösung noch in tatsächliche Pfade für jeden Tag umgewandelt werden. Dies hat lineare Laufzeit abhängig von der Anzahl der Knoten in den resultierenden Tagesstrecken. Insgesamt gibt es $m$ Kanten, die abgefahren werden müssen. Im schlimmsten Fall ist der kürzeste Weg zwischen jeder dieser $m$ Kanten $n - 1$ Kanten lang. Das heißt $\mathcal{O}(m \cdot n)$ lässt sich als Obergrenze für die Laufzeit feststellen. In der Praxis sind jedoch die kürzesten Wege zwischen den Kanten nicht immer $n - 1$ lang.

Auch für den 3. Teil des Algorithmus, das Minimieren von mehrfach abgefahrenen Kanten, muss die Lösung int tatsächliche Pfade umgewandelt werden, was Laufzeittechnische am aufwendigsten ist. Somit ist hier die Laufzeit auch $\mathcal{O}(m \cdot n)$.

#### 4.1.2 Speicherverbrauch

Zunächst einmal muss der Graph gespeichert werden. Diesen habe ich als Adjazenzliste gespeichert. Das heißt dieser verbraucht $\mathcal{O}(n + m)$ Speicherplatz. Theoretisch könnte $m$ bis zu $n^2$ groß werden, wenn es sich um einen vollständigen Graphen handelt. Das ist jedoch eher unwarscheinlich und auch die Beispiele sind davon weit entfernt.

Am meisten Speicherplatz verbrauchen die beiden Vectoren *distanz* und *parent*. Beide sind $n \cdot n$ groß, was Speicherverbrauch von $\mathcal{O}(n^2)$ entspricht. Wenn $\mathcal{O}(n^2)$ zu viel Speichervebrauch ist, könnte man alternativ auch das *distanz* und *parent* Array nicht speichern und die Werte jedesmal neu berechnen, wenn diese gebraucht werden. Das würde jedoch die Laufzeit verschlechtern.

### 4.2 Algorithmus 2

#### 4.2.1 Laufzeitanalyse

Zunächst werden die kürzesten Wege von der Zentrale zu allen Kanten berechnet. Diese Suche findet jedoch auf dem modifiziertem Graphen statt. Das heißt dieser Graph $n' = m + n$ Knoten und $m' = 2m$ Kanten. Allgemein befinden sich in der Prioritätswarteschlange des Dijkstra Algorithmus maximal $2m'$ Elemente. Jeder Knoten kann maximal so oft hinzugefügt werden, wie Kanten zu diesem führen. Da jede Kante genau zwei Knoten verbindet müssen es $2m'$ sein. Das heißt auch, dass über die Laufzeit des Programms insgesamt nur maximal $2m'$ Knoten der Prioritätswarteschlange hinzugefügt und entfernt werden. Jede Operation auf der Prioritätswarteschlange hat logarithmische Laufzeit. Das heißt die Laufzeit ist $\mathcal{O}(2m' \log 2m') \in \mathcal{O}(m' \log m')$. Mit $m' \in \mathcal{O}(m)$ und $m \in \mathcal{O}(n^2)$ lässt sich die Laufzeit noch beschränken auf $\mathcal{O}(m \log n)$.

Danach werden die $m$ Knoten, die originalen Kanten, zu denen die kürzeste Distanz berechnet wurde sortiert, was eine Laufzeit von $\mathcal{O}(m \log m) \in \mathcal{O}(m \log n)$ hat.

Hiernach werden die Pfade zu den originalen Kanten gespeichert. Da es insgesamt $m$ Kanten gibt und Kanten, die in Pfaden zu anderen Kanten enthalten sind nicht betrachtet werden dauert dies $\mathcal{O}(m)$.

Ich werden hier vernachlässigen, dass zuerst Kanten mit gleichem Endknoten verbunden werden, da dies auf jeden Fall schneller ist als der zweite Teil des Pfadeverbindens, da hier die Pfade noch die Kosten der Prioritätswarteschlange hinzukommen.

Als nächstes werden immer jeweils 2 Pfade verbunden. Die Laufzeit hierfür lässt sich nur schwer eingrenzen. Da es anfänglich maximal $m$ Pfade gibt, werden maximal $\mathcal{O}(m)$ Verbinden Operationen durchgeführt. Da die Pfade aller kürzeste Pfade sind und es keine negativen Kantengewichte gibt, sind diese Pfade maximal $n' - 1$ Kanten lang. Das Verbinden von zwei Pfaden läuft in lineare Zeit abhängig von der Gesamtlänge der beiden Pfade, die verbunden werden.

Nun ist die Frage, wie lang die Pfade werden, nachdem sie verbunden wurden. Im schlechtesten Fall haben sie nicht den gleichen Endpunkt. Ist das der Fall, wird aus zwei Pfaden mit Länge $(n' - 1)$ ein Pfad mit Länge $4(n' - 1)$, da sie zurück zur Zentrale geführt werden, sodass sie danach den gleichen Endknoten haben. Nach dieser Operation hat der Pfad jedoch die Zentrale als Endknoten. Das heißt nach dem ersten Durchlauf an Verbinden Operationen hat jeder Pfad den gleichen Endknoten. Das heißt, wenn jetzt wieder zwei Pfade verbunden werden, ist die Länge des resultierenden Pfades einfach die Summe der Längen der beiden ursprünglichen Pfade.

Im schlechtesten Fall werden also zunächst $\frac{m}{2}$ mal 2 Pfade mit Länge $n' - 1$ verbunden. Danach hat sich die Anzahl der Pfade halbiert und die Länge von jedem Pfad im schlimmsten Fall vervierfacht. Jetzt haben jedoch alle Pfade den gleichen Endknoten, sodass sich die Länge der Pfade nur noch verdoppelt. Danach werden also $\frac{m}{4}$ mal 2 Pfade mit Länge $4(n' - 1)$ verbunden. Und danach $\frac{m}{8}$ mal 2 Pfade mit Länge $8(n' - 1)$.
Der Einfachheit halber kann man sagen dies geht so weiter, bis es nur noch einen Pfad gibt (in Realität wird aufgehört, wenn noch 5 übrig sind). Da sich die Anzahl der Pfade mit jedem Durchlauf halbiert, werden maximal logarithmisch viele dieser Durchläufe durchgeführt. Das heißt im $i$-ten Durchlauf gibt es

$2^i$ Pfade wobei jeder Pfad maximal die Länge $2^i(n'-1)$ hat. Das heißt die Laufzeit lässt sich ausdrücken als

$$\sum_{i=1}^{\log m} \frac{m}{2^i} \cdot 2 \cdot 2^i(n'-1) = \sum_{i=1}^{\log m} 2m \cdot (n'-1) = 2m \cdot (n'-1) \cdot \log m$$

Da die Pfade durch eine Prioritätswarteschlange sortiert gehalten werden, kommt zu jedem dieser Operationen noch ein log-Faktor dazu:

$$2m \cdot (n'-1) \cdot \log^2 m \in \mathcal{O}(m \cdot n' \cdot \log^2 m)$$

mit $n' \in \mathcal{O}(m)$ und $m \in \mathcal{O}(n^2)$ kommt man auf $\mathcal{O}(m^2 \cdot \log^2 n)$

Jedoch spiegelt diese Oberenze nicht wirklich die tatsächliche Laufzeit des Programm wieder. In der Praxis terminiert das Programm bei allen Beispielen in unter 20 Millisekunden. Die Annahme, dass es $m$ Pfade gibt, die alle die Länge $n'-1$ haben, ist schon in der Praxis unmöglich. Wenn ein Pfad die Länge $n'-1$ hat, enthält dieser auch $n'-1$ Kanten, von denen die Hälfte originale Kanten sind, also gibt es auch dementsprechende weniger Pfade.

#### 4.2.2 Speicherverbrauch

Zunächst muss der modifizierte Graph mit $n' = n + m$ Knoten und $m' = 2m$ Kanten gespeichert werden. Da der Graph als Adjazenzliste gespeichert wird, braucht dies $\mathcal{O}(n'+m') \in \mathcal{O}(n+m)$Speicherplatz.

Zudem müssen noch die Pfade gespeichert werden. Wenn man auch hier wieder vom schlechtesten Fall ausgeht mit $m$ Pfaden der Länge $n'-1$ kommt man im schlimmtsten Fall mit $n'-1 \in \mathcal{O}(m)$ auf Speicherverbrauch von $\mathcal{O}(m^2)$.

## 5 Beispiele

Ich habe das Programm unter Manjaro Linux 5.10 programmiert und getestet. Das Programm nimmt zwei Kommandozeilenargumente. Zuerst eine Beispieldatei und als zweites entweder 1 oder 2, welcher Algorithmus ausgeführt werden soll.

Bei den Beispielen 0-4 habe ich für beide Algorithmen die komplette Ausgabe in die Dokumentation eingefügt. Bei den Beispielen 5-8 habe ich nur die Länge der gefundenen Tagesstrecken eingefügt. Die vollständigen Ausgaben für alle Beispiele von beiden Algorithmen befinden sich in meiner Einsendungs zip-Datei im Ordner Aufgabe1/Ausgaben.

### 5.1 Beispiel 0

Hier führe ich das Programm mit dem ersten Algorithmus aus. Das Programm terminiet nach ca. 0.05s. Die ausgegebende Anzahl an durchgeführten Iterationen bezieht sich auf die Iterationen des Hill Climbing Algorithmus.

```
./aufgabe1 muellabfuhr0.txt 1

Durchgefuehrte Iterationen: 39579
Tag 1: 0 -> 2 -> 0 -> 4 -> 0
Gesamtlaenge: 4
Tag 2: 0 -> 6 -> 0 -> 8 -> 7 -> 6 -> 0
Gesamtlaenge: 6
Tag 3: 0 -> 8 -> 1 -> 8 -> 0
Gesamtlaenge: 4
Tag 4: 0 -> 8 -> 9 -> 8 -> 1 -> 2 -> 0
Gesamtlaenge: 6
Tag 5: 0 -> 6 -> 5 -> 4 -> 3 -> 2 -> 0
Gesamtlaenge: 6
Maximale Laenge einer Tagestour: 6
```

Hier führe ich das Programm mit dem zweiten Algorithmus aus. Das Programm terminiert nach ca. 0.003s.

```
./aufgabe1 muellabfuhr0.txt 2

Tag 1: 0 -> 8 -> 9 -> 8 -> 0
Gesamtlaenge: 4
Tag 2: 0 -> 4 -> 5 -> 6 -> 0
Gesamtlaenge: 4
Tag 3: 0 -> 2 -> 1 -> 8 -> 0
Gesamtlaenge: 4
Tag 4: 0 -> 6 -> 7 -> 8 -> 0
Gesamtlaenge: 4
Tag 5: 0 -> 2 -> 3 -> 4 -> 0
Gesamtlaenge: 4
Maximale Laenge einer Tagestour: 4
```

## 5.2 Beispiel 1

Das Programm terminiert nach ca. 0.05s.

```
./aufgabe1 muellabfuhr1.txt 1

Durchgefuehrte Iterationen: 39615
Tag 1: 0 -> 6 -> 1 -> 3 -> 5 -> 7 -> 6 -> 0
Gesamtlaenge: 18
Tag 2: 0 -> 6 -> 3 -> 4 -> 0
Gesamtlaenge: 15
Tag 3: 0 -> 6 -> 7 -> 5 -> 4 -> 7 -> 6 -> 0
Gesamtlaenge: 19
Tag 4: 0 -> 5 -> 7 -> 6 -> 0
Gesamtlaenge: 10
Tag 5: 0 -> 6 -> 3 -> 2 -> 3 -> 6 -> 0
Gesamtlaenge: 18
Maximale Laenge einer Tagestour: 19
```

Das Programm terminiert nach ca. 0.003s.

```
./aufgabe1 muellabfuhr1.txt 2

Tag 1: 0 -> 6 -> 3 -> 2 -> 3 -> 6 -> 0
Gesamtlaenge: 18
Tag 2: 0 -> 6 -> 3 -> 5 -> 0
Gesamtlaenge: 11
Tag 3: 0 -> 6 -> 3 -> 1 -> 6 -> 0
Gesamtlaenge: 13
Tag 4: 0 -> 6 -> 3 -> 4 -> 0
Gesamtlaenge: 15
Tag 5: 0 -> 6 -> 7 -> 5 -> 4 -> 7 -> 6 -> 0
Gesamtlaenge: 19
Maximale Laenge einer Tagestour: 19
```

## 5.3 Beispiel 2

Das Programm terminiert nach ca. 0.06s.

```
./aufgabe1 muellabfuhr2.txt 1

Durchgefuehrte Iterationen: 39108
Tag 1: 0 -> 5 -> 0 -> 6 -> 9 -> 0 -> 9 -> 5 -> 9 -> 6 -> 9 -> 7 -> 9 -> 0
Gesamtlaenge: 13
Tag 2: 0 -> 9 -> 10 -> 9 -> 12 -> 9 -> 13 -> 1 -> 13 -> 3 -> 13 -> 4 -> 6 ->
       1 -> 6 -> 0
Gesamtlaenge: 15
Tag 3: 0 -> 6 -> 14 -> 10 -> 2 -> 8 -> 7 -> 11 -> 8 -> 12 -> 8 -> 11 -> 2 ->
```

```
      10 -> 9 -> 0
Gesamtlaenge: 15
Tag 4: 0 -> 6 -> 14 -> 13 -> 14 -> 2 -> 14 -> 5 -> 14 -> 7 -> 14 -> 8 ->
      11 -> 5 -> 0
Gesamtlaenge: 14
Tag 5: 0 -> 5 -> 11 -> 3 -> 11 -> 7 -> 1 -> 13 -> 4 -> 10 -> 4 -> 3 -> 13 ->
       1 -> 12 -> 9 -> 0
Gesamtlaenge: 16
Maximale Laenge einer Tagestour: 16
```

Das Programm terminiert nach ca. 0.004s.

```
./aufgabe1 muellabfuhr2.txt 2

Tag 1: 0 -> 5 -> 11 -> 2 -> 10 -> 9 -> 0 -> 9 -> 10 -> 4 -> 13 -> 9 -> 0
Gesamtlaenge: 12
Tag 2: 0 -> 5 -> 11 -> 3 -> 13 -> 9 -> 0 -> 6 -> 14 -> 7 -> 11 -> 5 -> 0
Gesamtlaenge: 12
Tag 3: 0 -> 6 -> 14 -> 8 -> 2 -> 14 -> 6 -> 0 -> 9 -> 5 -> 9 -> 0 -> 5 ->
      14 -> 5 -> 0
Gesamtlaenge: 15
Tag 4: 0 -> 9 -> 6 -> 9 -> 0 -> 6 -> 4 -> 3 -> 4 -> 6 -> 0 -> 9 -> 7 -> 1 ->
       6 -> 0 -> 9 -> 12 -> 8 -> 12 -> 9 -> 0
Gesamtlaenge: 21
Tag 5: 0 -> 9 -> 7 -> 8 -> 11 -> 5 -> 0 -> 9 -> 12 -> 1 -> 13 -> 9 -> 0 ->
       6 -> 14 -> 13 -> 14 -> 6 -> 0 -> 6 -> 14 -> 10 -> 14 -> 6 -> 0
Gesamtlaenge: 24
Maximale Laenge einer Tagestour: 24
```

## 5.4 Beispiel 3

Das Programm terminiert nach ca. 0.1s.

```
./aufgabe1 muellabfuhr3.txt 1

Durchgefuehrte Iterationen: 37851
Tag 1: 0 -> 1 -> 2 -> 0 -> 3 -> 4 -> 0 -> 5 -> 6 -> 0 -> 7 -> 8 -> 0 -> 9 ->
      10 -> 0 -> 11 -> 12 -> 0 -> 13 -> 14 -> 0 -> 14 -> 1 -> 14 -> 2 -> 14 ->
       3 -> 14 -> 4 -> 14 -> 5 -> 14 -> 6 -> 14 -> 7 -> 0
Gesamtlaenge: 36
Tag 2: 0 -> 7 -> 1 -> 7 -> 2 -> 7 -> 3 -> 7 -> 4 -> 7 -> 5 -> 7 -> 6 -> 1 ->
       6 -> 2 -> 6 -> 3 -> 6 -> 4 -> 5 -> 1 -> 5 -> 2 -> 5 -> 3 -> 5 -> 4 ->
       1 -> 4 -> 2 -> 3 -> 1 -> 3 -> 2 -> 0
Gesamtlaenge: 35
Tag 3: 0 -> 14 -> 8 -> 14 -> 9 -> 14 -> 10 -> 14 -> 11 -> 14 -> 12 -> 13 ->
       1 -> 13 -> 2 -> 13 -> 3 -> 13 -> 4 -> 13 -> 5 -> 13 -> 6 -> 13 -> 7 ->
      13 -> 8 -> 13 -> 9 -> 13 -> 10 -> 13 -> 11 -> 13 -> 12 -> 1 -> 12 -> 2 ->
      12 -> 3 -> 0
Gesamtlaenge: 40
Tag 4: 0 -> 12 -> 4 -> 12 -> 5 -> 12 -> 6 -> 12 -> 7 -> 12 -> 8 -> 12 -> 9 ->
      12 -> 10 -> 11 -> 1 -> 11 -> 2 -> 11 -> 3 -> 11 -> 4 -> 11 -> 5 -> 11 ->
       6 -> 11 -> 7 -> 11 -> 8 -> 11 -> 9 -> 11 -> 10 -> 1 -> 10 -> 2 -> 10 -> 3 -> 0
Gesamtlaenge: 40
Tag 5: 0 -> 10 -> 4 -> 10 -> 5 -> 10 -> 6 -> 10 -> 7 -> 10 -> 8 -> 9 -> 1 -> 9 ->
       2 -> 9 -> 3 -> 9 -> 4 -> 9 -> 5 -> 9 -> 6 -> 9 -> 7 -> 9 -> 8 -> 1 -> 8 ->
       2 -> 8 -> 3 -> 8 -> 4 -> 8 -> 5 -> 8 -> 6 -> 0
Gesamtlaenge: 38
Maximale Laenge einer Tagestour: 40
```

Das Programm terminiert nach ca. 0.004s.

```
./aufgabe1 muellabfuhr3.txt 2

Tag 1: 0 -> 11 -> 1 -> 14 -> 0 -> 13 -> 12 -> 14 -> 0 -> 13 -> 2 -> 12 -> 0 -> 12 ->
       5 -> 13 -> 0 -> 6 -> 4 -> 5 -> 0 -> 7 -> 4 -> 8 -> 0 -> 10 -> 9 -> 10 -> 0 ->
       14 -> 13 -> 14 -> 0
```

```
Gesamtlaenge: 32
Tag 2: 0 -> 4 -> 2 -> 3 -> 0 -> 3 -> 1 -> 4 -> 0 -> 10 -> 8 -> 9 -> 0 -> 14 -> 9 ->
      12 -> 0 -> 11 -> 7 -> 11 -> 0 -> 6 -> 5 -> 6 -> 0 -> 14 -> 11 -> 13 -> 0 -> 7 ->
      6 -> 8 -> 0
Gesamtlaenge: 32
Tag 3: 0 -> 10 -> 4 -> 9 -> 0 -> 11 -> 4 -> 12 -> 0 -> 9 -> 7 -> 8 -> 0 -> 14 -> 7 ->
      10 -> 0 -> 5 -> 3 -> 4 -> 0 -> 9 -> 2 -> 11 -> 0 -> 10 -> 5 -> 9 -> 0 -> 11 ->
      9 -> 13 -> 0
Gesamtlaenge: 32
Tag 4: 0 -> 7 -> 3 -> 7 -> 0 -> 12 -> 11 -> 12 -> 0 -> 13 -> 1 -> 12 -> 0 -> 12 ->
       7 -> 13 -> 0 -> 8 -> 1 -> 7 -> 0 -> 10 -> 1 -> 9 -> 0 -> 1 -> 2 -> 0 -> 12 ->
      10 -> 11 -> 0 -> 13 -> 6 -> 14 -> 0
Gesamtlaenge: 35
Tag 5: 0 -> 6 -> 2 -> 5 -> 0 -> 5 -> 1 -> 6 -> 0 -> 12 -> 6 -> 11 -> 0 -> 9 -> 6 ->
      10 -> 0 -> 12 -> 8 -> 11 -> 0 -> 14 -> 2 -> 10 -> 0 -> 8 -> 3 -> 6 -> 0 -> 13 ->
       4 -> 14 -> 0 -> 8 -> 2 -> 7 -> 0 -> 7 -> 5 -> 8 -> 0 -> 12 -> 3 -> 11 -> 0 ->
      13 -> 8 -> 14 -> 0 -> 14 -> 3 -> 13 -> 0 -> 9 -> 3 -> 10 -> 0 -> 11 -> 5 ->
      14 -> 0 -> 13 -> 10 -> 14 -> 0
Gesamtlaenge: 64
Maximale Laenge einer Tagestour: 64
```

## 5.5 Beispiel 4

Das Programm terminiert nach ca. 0.04s.

```
./aufgabe1 muellabfuhr4.txt 1

Durchgefuehrte Iterationen: 39633
Tag 1: 0 -> 1 -> 0 -> 9 -> 0
Gesamtlaenge: 4
Tag 2: 0 -> 9 -> 8 -> 7 -> 8 -> 9 -> 0
Gesamtlaenge: 6
Tag 3: 0 -> 1 -> 2 -> 3 -> 2 -> 1 -> 0
Gesamtlaenge: 6
Tag 4: 0 -> 9 -> 8 -> 7 -> 6 -> 5 -> 6 -> 7 -> 8 -> 9 -> 0
Gesamtlaenge: 10
Tag 5: 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 4 -> 3 -> 2 -> 1 -> 0
Gesamtlaenge: 10
Maximale Laenge einer Tagestour: 10
```

Das Programm terminiert nach ca. 0.003s.

```
./aufgabe1 muellabfuhr4.txt 2

Tag 1: 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 0
Gesamtlaenge: 10
Tag 2:
Gesamtlaenge: 0
Tag 3:
Gesamtlaenge: 0
Tag 4:
Gesamtlaenge: 0
Tag 5:
Gesamtlaenge: 0
Maximale Laenge einer Tagestour: 10
```

## 5.6 Beispiel 5

Das Programm terminiert nach ca. 0.5s.

```
./aufgabe1 muellabfuhr5.txt 1

Durchgefuehrte Iterationen: 24802
Tag 1: [...]
```

```
Gesamtlaenge: 1742
Tag 2: [...]
Gesamtlaenge: 1798
Tag 3: [...]
Gesamtlaenge: 1762
Tag 4: [...]
Gesamtlaenge: 1777
Tag 5: [...]
Gesamtlaenge: 1758
Maximale Laenge einer Tagestour: 1798
```

Das Programm terminiert nach ca. 0.007s.

```
./aufgabe1 muellabfuhr5.txt 2

Tag 1: [...]
Gesamtlaenge: 1502
Tag 2: [...]
Gesamtlaenge: 1680
Tag 3: [...]
Gesamtlaenge: 1860
Tag 4: [...]
Gesamtlaenge: 2071
Tag 5: [...]
Gesamtlaenge: 2490
Maximale Laenge einer Tagestour: 2490
```

## 5.7 Beispiel 6

Das Programm terminiert nach ca. 0.17s.

```
./aufgabe1 muellabfuhr6.txt 1

Durchgefuehrte Iterationen: 34779
Tag 1: [...]
Gesamtlaenge: 934913
Tag 2: [...]
Gesamtlaenge: 652524
Tag 3: [...]
Gesamtlaenge: 841135
Tag 4: [...]
Gesamtlaenge: 909217
Tag 5: [...]
Gesamtlaenge: 479374
Maximale Laenge einer Tagestour: 934913
```

Das Programm terminiert nach ca. 0.004s.

```
./aufgabe1 muellabfuhr6.txt 2

Tag 1: [...]
Gesamtlaenge: 1505895
Tag 2: [...]
Gesamtlaenge: 1597911
Tag 3: [...]
Gesamtlaenge: 1753919
Tag 4: [...]
Gesamtlaenge: 2108521
Tag 5: [...]
Gesamtlaenge: 2593283
Maximale Laenge einer Tagestour: 2593283
```

## 5.8 Beispiel 7

Das Programm terminiert nach ca. 0.7s.

```
./aufgabe1 muellabfuhr7.txt 1

Durchgefuehrte Iterationen: 14974
Tag 1: [...]
Gesamtlaenge: 1470383
Tag 2: [...]
Gesamtlaenge: 1162790
Tag 3: [...]
Gesamtlaenge: 1233637
Tag 4: [...]
Gesamtlaenge: 895341
Tag 5: [...]
Gesamtlaenge: 839122
Maximale Laenge einer Tagestour: 1470383
```

Das Programm terminiert nach ca. 0.012s.

```
./aufgabe1 muellabfuhr7.txt 2

Tag 1: [...]
Gesamtlaenge: 33900780
Tag 2: [...]
Gesamtlaenge: 38035399
Tag 3: [...]
Gesamtlaenge: 41814152
Tag 4: [...]
Gesamtlaenge: 46692723
Tag 5: [...]
Gesamtlaenge: 58776235
Maximale Laenge einer Tagestour: 58776235
```

## 5.9 Beispiel 8

Das Programm terminiert nach ca. 0.8s.

```
./aufgabe1 muellabfuhr8.txt 1

Durchgefuehrte Iterationen: 5157
Tag 1: [...]
Gesamtlaenge: 4956463
Tag 2: [...]
Gesamtlaenge: 4578770
Tag 3: [...]
Gesamtlaenge: 4746011
Tag 4: [...]
Gesamtlaenge: 4236841
Tag 5: [...]
Gesamtlaenge: 4687578
Maximale Laenge einer Tagestour: 4956463
```

Das Programm terminiert nach ca. 0.035s.

```
./aufgabe1 muellabfuhr8.txt 2

Tag 1: [...]
Gesamtlaenge: 26994078
Tag 2: [...]
Gesamtlaenge: 30549478
Tag 3: [...]
```

```
Gesamtlaenge: 35828505
Tag 4: [...]
Gesamtlaenge: 41242037
Tag 5: [...]
Gesamtlaenge: 48178252
Maximale Laenge einer Tagestour: 48178252
```

# 6 Quellcode

## 6.1 Algorithmus 1

### 6.1.1 Generierung der Ausgangslösung

```
/**
 * Generiert eine Anfangsloesung fuer den Hill-Climbing Algorithmus.
 * Kanten, die nahe beieinander liegen werden eher demselben Tag zugeordnet
 * @param kanten Liste aller Kanten des Graphen
 * @return Gibt die Anfangsloesung als loesung zurueck
 */
loesung HillClimbing::genAnfangsLoesung(
            const std::vector<std::tuple<int, int, int>> &kanten) {

    // Adjazenzliste des Graphen
    std::vector<std::vector<std::tuple<int, int, int, int>>> adjazenzListe(n);

    int index = n;
    for (std::tuple<int, int, int> tpl: kanten) {
        int a = std::get<0>(tpl);
        int b = std::get<1>(tpl);
        int l = std::get<2>(tpl);
        auto tup = std::make_tuple(a, b, l, index);

        adjazenzListe[a].emplace_back(tup);
        adjazenzListe[b].emplace_back(tup);
        kantenIndizes[std::min(a, b)][std::max(a, b)] = index;
        kantenSpeicher[index - n] = std::make_tuple(a, b, l);
        index++;
    }

    loesung ls;
    ls.resize(5);

    // Speichert, ob ein Knoten bereits dem Stack hinzugefuegt wurde
    std::vector<bool> onStack(n);
    std::stack<int> s;
    s.push(0);

    // Speichert, welche Kanten bereits einem Tag zugewiesen wurden
    std::vector<bool> zugewiesen(m);

    int tagIndex = 0; // Tag welchem momentan Kanten zugewiesen werden
    const int proTag = m / 5; // Wie viele Kanten jeder Tag zugeordnet bekommt

    // Wie viele Kanten dem momentanen Tag bereits zugeordnet wurden
    int anzahlZugewiesen = 0;

    while (!s.empty()) {

        int knoten = s.top();
        s.pop();

        // Gehe alle Nachbarknoten durch
        for (std::tuple<int, int, int, int> tpl: adjazenzListe[knoten]) {
            int a = std::get<0>(tpl);
            int b = std::get<1>(tpl);
            int ind = std::get<3>(tpl);

            // ind geht von n bis m + n - 1, daher muss n abgezogen werden
            if (zugewiesen[ind - n]) continue; //Kante wurde bereits einem Tag zugewiesen
```

```
            zugewiesen[ind - n] = true;

            if (anzahlZugewiesen >= proTag && tagIndex != 4) {
                // Gehe zu naechstem Tag
                anzahlZugewiesen = 0;
                tagIndex++;
            }

            // Speichere Kante
            ls[tagIndex].emplace_back(ind);
            anzahlZugewiesen++;

            // Fuege Knoten Stack hinzu, wenn nicht schon geschehen
            if (!onStack[a]) {
                s.push(a);
                onStack[a] = true;
            }
            if (!onStack[b]) {
                s.push(b);
                onStack[b] = true;
            }
        }

    }

    return ls;
}
```

### 6.1.2 Hill Climbing Algorithmus

```

typedef std::vector<std::vector<int>> loesung;
#define ZENTRALE 0

/**
 * Fuehrt den zweiten Algorithmus aus
 * @param kanten Eine Liste aller Kanten des Graphen
 */
void HillClimbing::run(std::vector<std::tuple<int, int, int>> &kanten) {

    loesung ls = genAnfangsLoesung(kanten);

    const long double alpha = 0.00046;
    const long long iterationen = 40000 * std::exp(-(n + m) * alpha);
    long long besterWert = berechneWert(ls);

    for (int i = 0; i < iterationen; ++i) {
        loesung neueLoesung = berechneNachbar(ls);

        // Berechne Wert der neuen Loesung
        long long neuerWert = berechneWert(neueLoesung);

        if (neuerWert < besterWert) {
            // Aktualisiere beste Loesung
            besterWert = neuerWert;
            ls = neueLoesung;
        }
    }

    ls = entferneMehrfachAbgefahreneStrassen(ls);

    // Berechne Laenge fuer alle Tagesstrecken
    std::vector<long long> laenge(5);
    long long wert = 0;
    for (int i = 0; i < 5; ++i) {
        laenge[i] = berechneTagesstrecke(ls[i]);
        wert = std::max(wert, laenge[i]);

        // Konvertiere Loesung in richtige Pfade im Graphen
        ls[i] = konvertiereZuNormal(ls[i]);
```

```
    }

    // IO
    // [...]
}

/**
 * Berechnet einen Nachbarn der gegebenen Loesung im Suchraum.
 * @param s Eine Loesung
 * @return Gibt die gegebene Loesung leicht veraendert zurueck
 */
loesung HillClimbing::berechneNachbar(loesung s) {

    // Bestimme Tag mit laengster Strecke
    int maxIndex = 0;
    long long maxWert = 0;
    for (int i = 0; i < 5; ++i) {
        long long val = berechneTagesstrecke(s[i]);
        if (val > maxWert) {
            maxWert = val;
            maxIndex = i;
        }
    }

    // Tausche Kante zwischen zwei Tagen
    int tag1 = maxIndex;
    // Bestimme zufaellig einen anderen Tag zum Tauschen
    unsigned int tag2 = tag1;
    while (tag2 == tag1) {
        tag2 = zufaelligeZahl(0, 4);
    }

    if (!s[tag1].empty()) {
        unsigned int index = zufaelligeZahl(0, s[tag1].size() - 1);
        int strasse = s[tag1][index];
        s[tag1].erase(s[tag1].begin() + index);
        s[tag2].emplace_back(strasse);
    }

    // Tausche Reihenfolge der Strassen an einem Tag
    if (s[tag1].size() >= 2) {
        unsigned int index1 = zufaelligeZahl(0, s[tag1].size() - 1);
        unsigned int index2 = zufaelligeZahl(0, s[tag1].size() - 1);

        std::swap(s[tag1][index1], s[tag1][index2]);
    }

    return s;
}

/**
 * Berechnet den Wert einer Loesung. Gibt die Laenge der laengsten Tagesstrecke
 * zurueck
 * @param s Eine Loesung
 */
long long HillClimbing::berechneWert(const loesung &s) { // O(m)
    long long maxTotal = 0;

    for (const auto &pfad: s) {
        maxTotal = std::max(maxTotal, berechneTagesstrecke(pfad));
    }

    return maxTotal;
}

/**
 * Berechnet die Laenge einer Tagesstrecke
 * @param tagesstrecke Eine Liste an Kanten
 * @return Gibt die Laenge des kuerzesten Pfades zurueck, der die Kanten in der gegebenen
 *         Reihenfolge abfaehrt
 */
long long HillClimbing::berechneTagesstrecke(const std::vector<int> &tagesstrecke) {
    int vorherigerKnoten = ZENTRALE;
```

```
    long long gesamtStrecke = 0;

    for (int kante: tagesstrecke) {
        auto r = berechneStrecke(vorherigerKnoten, kante);
        gesamtStrecke += r.first;
        vorherigerKnoten = r.second;
    }

    // Strecke zurueck zur Zentrale addieren
    gesamtStrecke += berechneStrecke(vorherigerKnoten, ZENTRALE).first;

    return gesamtStrecke;
}

/**
 * Berechnet die Entfernung zwischen den Knoten s und t
 * @param s Der Startknoten, muss ein originaler Knoten sein
 * @param t Der Zielknoten
 * @return Gibt die Laenge der Strecke zurueck und den originalen Knoten, an dem man sich
 *         danach befindet.
 */
std::pair<long long, int> HillClimbing::berechneStrecke(int s, int t) {
    // Stelle sicher, dass s ein echter Knoten ist
    if (t == s) {
        return {0, s};
    }

    int knoten1 = t;
    int knoten2 = t;

    if (t >= n) {
        auto k1 = get<0>(kantenSpeicher[t - n]);
        auto k2 = get<1>(kantenSpeicher[t - n]);

        if (distanz[s][k1] < distanz[s][k2]) {
            knoten1 = k1;
            knoten2 = k2;
        } else {
            knoten1 = k2;
            knoten2 = k1;
        }
    }

    return {distanz[s][knoten1] + adjMatrix[knoten1][knoten2], knoten2};
}
```

#### 6.1.3 Versuch mehrfaches Abfahren von Straßen zu vermeiden

```
/**
 * Versucht zu minimieren, dass Kanten unnoetigerweise mehrfach abgefahren werden
 * @param s Eine Loesung
 * @return Gibt die verbesserte Loesung zurueck
 */
loesung HillClimbing::entferneMehrfachAbgefahreneStrassen(loesung s) {
    std::unordered_map<int, long long> tagesstrecken;
    tagesstrecken.reserve(10);
    for (int i = 0; i < 5; ++i) { // O(m)
        if (s[i].empty()) continue;
        // Nutze erste Kante in s[i] als Key, da jede Kante nur einem Tag zugeordnet ist
        // -> s[i][0] ist eindeutig
        tagesstrecken[s[i][0]] = berechneTagesstrecke(s[i]);
    }

    // Sortiere nach aufsteigender Laenge
    std::sort(s.begin(), s.end(), [&](auto &a, auto &b) {
        if (a.empty()) {
            return true;
        } else if (b.empty()) {
            return false;
        }
```

```
        return tagesstrecken[a[0]] < tagesstrecken[b[0]];
    });

    for (int i = 0; i < 5; ++i) {
        std::vector<bool> abgefahren(m);

        std::vector<int> normal = konvertiereZuNormal(s[i]);
        // Speichere welche Kanten in Tagesstrecke i abgefahren werden
        for (int k = 1; k < normal.size(); ++k) {
            int a = normal[k - 1];
            int b = normal[k];
            int index = kantenIndizes[std::min(a, b)][std::max(a, b)];
            // Index geht von n bis n + m - 1
            abgefahren[index - n] = true;
        }

        // Entferne die gefundenen Kanten aus den anderen Tagesstrecken
        for (int j = i + 1; j < 5; ++j) {
            for (int k = 0; k < s[j].size(); ++k) {
                if (abgefahren[s[j][k] - n]) {
                    // Markieren fuer spaetere Loeschung
                    s[j][k] = -1;
                }
            }
            // Loesche zuvor markierte Elemente
            s[j].erase(std::remove(s[j].begin(), s[j].end(), -1), s[j].end());
        }
    }
    return s;
}

/**
 * Wandelt die gegebene Liste an Kanten in einem richtigen Pfad im Graphen um
 * @param tagesstrecke Eine Liste an Kanten
 * @return Gibt ein Liste an zusammenhaengenden Knoten zurueck, die die in tagesstrecke
 *         gegebenen Kanten enthaelt
 */
std::vector<int>
HillClimbing::konvertiereZuNormal(std::vector<int> tagesstrecke) {

    std::vector<int> normalerPfad;
    int vorherigerKnoten = 0;

    tagesstrecke.emplace_back(ZENTRALE); // Pfad muss bei der Zentrale enden
    for (int kante: tagesstrecke) {
        int ziel = kante;

        // Ueberprueft, ob "kante" wirklich eine Kante ist oder ein Knoten
        if (kante >= n) {
            // ist eine Kante

            // Die beiden Knoten, die durch die momentane Kante verbunden sind
            auto knoten1 = get<0>(kantenSpeicher[kante - n]);
            auto knoten2 = get<1>(kantenSpeicher[kante - n]);

            // Bestimme, welcher der beiden Knoten naeher an momentaner Position ist
            if (distanz[vorherigerKnoten][knoten1] <
                        distanz[vorherigerKnoten][knoten2]) {
                ziel = knoten1;
            } else {
                ziel = knoten2;
            }
        }

        normalerPfad.emplace_back(vorherigerKnoten);

        // Bestimme und speichere Pfad von vorherigerKnoten zu ziel
        do {
            vorherigerKnoten = parent[vorherigerKnoten][ziel];
            if (vorherigerKnoten == normalerPfad[normalerPfad.size() - 1]) {
                // Wurde bereits gespeichert -> ueberspringen
                break;
```

```
            }
            normalerPfad.emplace_back(vorherigerKnoten);
        } while (vorherigerKnoten != ziel);

        // Wenn momentane Position eine Kante ist, muss die momentane
        // Position noch aktualisiert werden
        if (kante >= n) {
            // Aktualisiere momentane Position

            // Die beiden Knoten, die durch die momentane Kante verbunden sind
            auto node1 = get<0>(kantenSpeicher[kante - n]);
            auto node2 = get<1>(kantenSpeicher[kante - n]);

            if (ziel == node1) {
                ziel = node2;
            } else {
                ziel = node1;
            }

            vorherigerKnoten = ziel;
        }

    }

    return normalerPfad;
}
```

## 6.2 Algorithmus 2

### 6.2.1 Berechnen der Pfade

```
/**
 * Berechnet die kuerzesten Wege zu allen Kanten des originalen Graphen
 * @return Gibt die Pfade zu den Kanten mit Laenge zurueck
 */
std::vector<std::pair<std::vector<int>, long long>> Algorithmus2::berechnePfade() {

    // parent[v] speichert den naechten Knoten auf dem kuerzesten Pfad von v zur Zentrale
    std::vector<std::pair<int, int>> parent(graph.size());

    std::priority_queue<std::pair<int, int>> pq;
    std::vector<bool> visited(graph.size());

    // Speichert die Distanz von der Zentrale zu den Knoten
    std::vector<int> dist(graph.size(), INT32_MAX);

    dist[0] = 0;
    pq.push(std::make_pair(0, 0));

    while (!pq.empty()) {
        auto top = pq.top();
        pq.pop();

        int knoten = top.second;
        if (visited[knoten]) continue;
        visited[knoten] = true;

        for (auto adj: graph[knoten]) {
            int adjKnoten = adj.first;
            int kantenGewicht = adj.second;

            if (dist[knoten] + kantenGewicht < dist[adjKnoten]) {
                dist[adjKnoten] = dist[knoten] + kantenGewicht;
                pq.push(std::make_pair(dist[adjKnoten] * (-1), adjKnoten));
                parent[adjKnoten] = std::make_pair(knoten, kantenGewicht);
            }
        }
    }
```

```
    std::vector<std::pair<std::vector<int>, long long>> pfade;
    pfade.reserve(m);


    // Liste mit Indizes der Kanten, um sie zu sortieren
    std::vector<int> kanten(m);
    for (int i = n; i < n + m; ++i) {
        kanten[i - n] = i;
    }

    // Sortiere Liste mit Indizes der Kanten nach aufsteigender Entfernung zur Zentrale
    std::sort(kanten.begin(), kanten.end(), [&](int a, int b) {
        return dist[a] > dist[b];
    });

    // Speichert, welche Kanten bereits auf dem Weg zu anderen Kanten abgefahren wurde
    // und somit nicht weiter betrachtet werden muessen
    std::vector<int> bereitsAbgefahren(n + m);

    for (int i: kanten) { // O(m)
        if (bereitsAbgefahren[i] > 0) continue;
        int kante = i;
        std::pair<int, std::pair<int, int>> tmp;

        for (auto adj: graph[kante]) {
            if (adj.first != parent[kante].first) {
                tmp.first = adj.first;
                tmp.second = parent[adj.first];
                parent[adj.first] = std::make_pair(kante, adj.second);
                kante = adj.first;
                break;
            }
        }

        std::vector<int> pfad;
        pfad.reserve(graph.size());
        int kosten = parent[kante].second;
        int par = kante;
        pfad.push_back(par);
        if (par >= n) {
            bereitsAbgefahren[par]++;
        }

        do {
            par = parent[par].first;
            pfad.push_back(par);
            if (par >= n) {
                bereitsAbgefahren[par]++;
            } else {
                // Nur originale Kanten addieren zur Gesamtlaenge des Pfades
                kosten += parent[par].second;
            }
        } while (par != ZENTRALE);

        // Drehe Pfad um damit er bei der Zentrale beginnt
        std::reverse(pfad.begin(), pfad.end());
        pfade.emplace_back(std::make_pair(pfad, kosten));

        // Mache Aenderungen an parent vector wieder rueckgaengig
        parent[tmp.first] = tmp.second;
    }

    return pfade;
}
```

### 6.2.2 Verbinden der Pfade

```
/**
 * Verbindet die gegebenen Pfade solange, bis nur noch 5 uebrig sind
```

```
 * @param pfade Eine Liste an Pfaden
 * @return Gibt eine Liste mit den verbleibenden 5 Pfaden zurueck
 */
std::vector<std::pair<std::vector<int>, long long>> Algorithmus2::verbindePfade(
        std::vector<std::pair<std::vector<int>, long long>> &pfade) {

    // Speichert fuer jeden Endknoten, welche Pfaden auf diesen enden
    std::unordered_map<int, std::vector<int>> endKnoten;
    endKnoten.reserve(pfade.size() * 2);

    // Speichere fuer jeden Pfad den Endknoten
    for (int i = 0; i < pfade.size(); ++i) {
        int letzterKnoten = pfade[i].first[pfade[i].first.size() - 1];

        // Speichere Index des Pfades
        endKnoten[letzterKnoten].emplace_back(i);
    }

    // Halte Pfade sortiert nach ihrer Laenge
    auto compare = [](auto &a, auto &b) {
        return a.second > b.second;
    };
    std::priority_queue<std::pair<std::vector<int>, long long>,
            std::vector<std::pair<std::vector<int>, long long>>, decltype(compare)> pq;


    // Verbinde jeweils zwei Pfade mit gleichem Endknoten.
    for (auto &pair: endKnoten) {
        if (pair.second.size() < 2) {
            // Weniger als 2 Pfade mit diesem Endknoten
            // wird uebersprungen
            if (pair.second.empty()) continue;
            pq.push(pfade[pair.second[0]]);
            continue;
        }

        if (pair.second.size() % 2 != 0) {
            // Anzahl ist nicht durch 2 teilbar. Ein Pfad kann nicht verbunden
            // werden und wird uebersprungen.
            pq.push(pfade[pair.second[pair.second.size() - 1]]);
            pair.second.pop_back();
        }

        for (int i = 0; i < pair.second.size() - 1; i += 2) {
            // Verbinde jeweils zwei Pfade
            verbinde(pfade[pair.second[i]], pfade[pair.second[i + 1]]);
            pq.push(pfade[pair.second[i]]);
        }
    }

    while (pq.size() > 5) {

        // Nehme die zwei Pfade mit der kuerzesten Gesamtstrecke
        auto pfad1 = pq.top();
        pq.pop();
        auto pfad2 = pq.top();
        pq.pop();

        // Verbinde beide Pfade
        verbinde(pfad1, pfad2);

        // Ergebnis wieder in Priority Queue speichern
        pq.push(pfad1);
    }

    pfade.clear();
    pfade.resize(5);

    unsigned int size = pq.size();
    for (int i = 0; i < size; ++i) {
        pfade[i] = pq.top();
        pq.pop();
    }
```

```
    return pfade;
}
```