

Aufgabe 1: Müllabfuhr

Teilnahme-ID: 60963

Tobias Hettler

24. April 2022

Inhaltsverzeichnis

1 Komplexität	2
2 Algorithmus 1	2
2.1 Lösungsidee	2
2.1.1 Repräsentation	2
2.1.2 Generierung der Ausgangslösung	3
2.1.3 Hill Climbing	3
2.1.4 Versuch das mehrfache Abfahren von Straßen zu minimieren	5
2.2 Umsetzung	5
2.2.1 Repräsentation einer Lösung	5
2.2.2 Berechnung von $d(u, v)$	5
2.2.3 Wert einer Lösung berechnen	5
2.2.4 Umwandeln der Lösung	5
3 Algorithmus 2	6
3.1 Lösungsidee	6
3.1.1 Repräsentation	6
3.1.2 Berechnung der Pfade	6
3.2 Durchführung	6
3.2.1 Repräsentation	6
3.2.2 Berechnung der Pfade	7
3.2.3 Verbinden der Pfade	7
4 Komplexitätsanalyse	7
4.1 Algorithmus 1	7
4.1.1 Laufzeitanalyse	7
4.1.2 Speicherverbrauch	8
4.2 Algorithmus 2	8
4.2.1 Laufzeitanalyse	8
4.2.2 Speicherverbrauch	9
5 Beispiele	9
5.1 Beispiel 0	9
5.2 Beispiel 1	10
5.3 Beispiel 2	10
5.4 Beispiel 3	11
5.5 Beispiel 4	12
5.6 Beispiel 5	12
5.7 Beispiel 6	13
5.8 Beispiel 7	14
5.9 Beispiel 8	14

6 Quellcode	15
6.1 Algorithmus 1	15
6.1.1 Generierung der Ausgangslösung	15
6.1.2 Hill Climbing Algorithmus	16
6.1.3 Versuch mehrfaches Abfahren von Straßen zu vermeiden	18
6.2 Algorithmus 2	20
6.2.1 Berechnen der Pfade	20
6.2.2 Verbinden der Pfade	21

1 Komplexität

Theorem 1.1. Das Müllabfuhr Problem ist NP-Schwer.

Beweis. Um dies zu beweisen, reduziere ich vom multiway number partitioning Problem¹, 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 partitioning 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 partitioning 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.

¹https://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

```

1 Stack s // Stack nach FIFO Prinzip
2 while s nicht leer do
3   v ← s.top() // Nehme obersten Knoten vom Stack
4   forall Nachbarn u von v do
5     if Kante v - u noch nicht zugewiesen then
6       // Wenn Tag i bereits  $\lfloor \frac{m}{5} \rfloor$  Straßen zugeordnet sind, gehe zu Tag i + 1
7       Weise Kante v - u Tag i zu
8     end if
9     if Knoten u unbekannt then
10      // Füge Knoten u dem Stack hinzu
11      s.push(u)
12      markiere u als bekannt
13    end if
14  end forall
15 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 α 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 Tagesstrecke zu verändern.

Algorithmus 2 : Hill Climbing Algorithmus

```

1 Loesung  $s$  // Ausgangslösung, wie oben beschrieben generiert
2 for  $i \leftarrow 1, \text{iterationen}$  do
3   |  $s' \leftarrow \text{nachbar}(s)$ 
4   | if  $s'$  besser als  $s$  then
5     | |  $s \leftarrow s'$ 
6   | end if
7 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:

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

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

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

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 aneinander, 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:

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

Es muss noch $d(T_n, 0)$ addiert werden, da jede Tagestrecke 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 Tagestrecke 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 Tagesstrecken mit Index $j > i$ die Kanten gelöscht, die bei i „zwischendrin“ abgefahren werden.

Dadurch, dass die Tagesstrecken sortiert wurden, werden die Chancen, dass 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 ursprünglichen 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 beliebigen 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 anfängliche 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 Gesamlaufzeit $\mathcal{O}(n^2 \log n)$.

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 in 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 unwahrscheinlich 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 Speicherverbrauch 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 modifizierten 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 werde 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 Oberenzen nicht wirklich die tatsächliche Laufzeit des Programms 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 dementsprechend 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 schlimmsten 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 Einsendungszip-Datei im Ordner Aufgabe1/Ausgaben.

5.1 Beispiel 0

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

```

1 ./aufgabe1 muellabfuhr0.txt 1

3 Durchgefuehrte Iterationen: 39579
4 Tag 1: 0 -> 2 -> 0 -> 4 -> 0
5 Gesamtlaenge: 4
6 Tag 2: 0 -> 6 -> 0 -> 8 -> 7 -> 6 -> 0
7 Gesamtlaenge: 6
8 Tag 3: 0 -> 8 -> 1 -> 8 -> 0
9 Gesamtlaenge: 4
10 Tag 4: 0 -> 8 -> 9 -> 8 -> 1 -> 2 -> 0
11 Gesamtlaenge: 6
12 Tag 5: 0 -> 6 -> 5 -> 4 -> 3 -> 2 -> 0
13 Gesamtlaenge: 6
14 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
2
  Tag 1: 0 -> 8 -> 9 -> 8 -> 0
4 Gesamtlaenge: 4
  Tag 2: 0 -> 4 -> 5 -> 6 -> 0
6 Gesamtlaenge: 4
  Tag 3: 0 -> 2 -> 1 -> 8 -> 0
8 Gesamtlaenge: 4
  Tag 4: 0 -> 6 -> 7 -> 8 -> 0
10 Gesamtlaenge: 4
  Tag 5: 0 -> 2 -> 3 -> 4 -> 0
12 Gesamtlaenge: 4
  Maximale Laenge einer Tagestour: 4
```

5.2 Beispiel 1

Das Programm terminiert nach ca. 0.05s.

```
./aufgabe1 muellabfuhr1.txt 1
3 Durchgefuehrte Iterationen: 39615
  Tag 1: 0 -> 6 -> 1 -> 3 -> 5 -> 7 -> 6 -> 0
5 Gesamtlaenge: 18
  Tag 2: 0 -> 6 -> 3 -> 4 -> 0
7 Gesamtlaenge: 15
  Tag 3: 0 -> 6 -> 7 -> 5 -> 4 -> 7 -> 6 -> 0
9 Gesamtlaenge: 19
  Tag 4: 0 -> 5 -> 7 -> 6 -> 0
11 Gesamtlaenge: 10
  Tag 5: 0 -> 6 -> 3 -> 2 -> 3 -> 6 -> 0
13 Gesamtlaenge: 18
  Maximale Laenge einer Tagestour: 19
```

Das Programm terminiert nach ca. 0.003s.

```
./aufgabe1 muellabfuhr1.txt 2
2
  Tag 1: 0 -> 6 -> 3 -> 2 -> 3 -> 6 -> 0
4 Gesamtlaenge: 18
  Tag 2: 0 -> 6 -> 3 -> 5 -> 0
6 Gesamtlaenge: 11
  Tag 3: 0 -> 6 -> 3 -> 1 -> 6 -> 0
8 Gesamtlaenge: 13
  Tag 4: 0 -> 6 -> 3 -> 4 -> 0
10 Gesamtlaenge: 15
  Tag 5: 0 -> 6 -> 7 -> 5 -> 4 -> 7 -> 6 -> 0
12 Gesamtlaenge: 19
  Maximale Laenge einer Tagestour: 19
```

5.3 Beispiel 2

Das Programm terminiert nach ca. 0.06s.

```
./aufgabe1 muellabfuhr2.txt 1
3 Durchgefuehrte Iterationen: 39108
  Tag 1: 0 -> 5 -> 0 -> 6 -> 9 -> 0 -> 9 -> 5 -> 9 -> 6 -> 9 -> 7 -> 9 -> 0
5 Gesamtlaenge: 13
  Tag 2: 0 -> 9 -> 10 -> 9 -> 12 -> 9 -> 13 -> 1 -> 13 -> 3 -> 13 -> 4 -> 6 ->
7 1 -> 6 -> 0
  Gesamtlaenge: 15
9 Tag 3: 0 -> 6 -> 14 -> 10 -> 2 -> 8 -> 7 -> 11 -> 8 -> 12 -> 8 -> 11 -> 2 ->
```

```

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

```

Das Programm terminiert nach ca. 0.004s.

```

./aufgabe1 muellabfuhr2.txt 2
2
Tag 1: 0 -> 5 -> 11 -> 2 -> 10 -> 9 -> 0 -> 9 -> 10 -> 4 -> 13 -> 9 -> 0
4 Gesamtlaenge: 12
Tag 2: 0 -> 5 -> 11 -> 3 -> 13 -> 9 -> 0 -> 6 -> 14 -> 7 -> 11 -> 5 -> 0
6 Gesamtlaenge: 12
Tag 3: 0 -> 6 -> 14 -> 8 -> 2 -> 14 -> 6 -> 0 -> 9 -> 5 -> 9 -> 0 -> 5 ->
8 14 -> 5 -> 0
Gesamtlaenge: 15
10 Tag 4: 0 -> 9 -> 6 -> 9 -> 0 -> 6 -> 4 -> 3 -> 4 -> 6 -> 0 -> 9 -> 7 -> 1 ->
6 -> 0 -> 9 -> 12 -> 8 -> 12 -> 9 -> 0
12 Gesamtlaenge: 21
Tag 5: 0 -> 9 -> 7 -> 8 -> 11 -> 5 -> 0 -> 9 -> 12 -> 1 -> 13 -> 9 -> 0 ->
14 6 -> 14 -> 13 -> 14 -> 6 -> 0 -> 6 -> 14 -> 10 -> 14 -> 6 -> 0
Gesamtlaenge: 24
16 Maximale Laenge einer Tagestour: 24

```

5.4 Beispiel 3

Das Programm terminiert nach ca. 0.1s.

```

./aufgabe1 muellabfuhr3.txt 1
2
Durchgefuehrte Iterationen: 37851
4 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 ->
6 3 -> 14 -> 4 -> 14 -> 5 -> 14 -> 6 -> 14 -> 7 -> 0
Gesamtlaenge: 36
8 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 ->
10 1 -> 4 -> 2 -> 3 -> 1 -> 3 -> 2 -> 0
Gesamtlaenge: 35
12 Tag 3: 0 -> 14 -> 8 -> 14 -> 9 -> 14 -> 10 -> 14 -> 11 -> 14 -> 12 -> 13 ->
1 1 -> 13 -> 2 -> 13 -> 3 -> 13 -> 4 -> 13 -> 5 -> 13 -> 6 -> 13 -> 7 ->
14 13 -> 8 -> 13 -> 9 -> 13 -> 10 -> 13 -> 11 -> 13 -> 12 -> 1 -> 12 -> 2 ->
12 12 -> 3 -> 0
16 Gesamtlaenge: 40
Tag 4: 0 -> 12 -> 4 -> 12 -> 5 -> 12 -> 6 -> 12 -> 7 -> 12 -> 8 -> 12 -> 9 ->
18 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
20 Gesamtlaenge: 40
Tag 5: 0 -> 10 -> 4 -> 10 -> 5 -> 10 -> 6 -> 10 -> 7 -> 10 -> 8 -> 9 -> 1 -> 9 ->
22 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
24 Gesamtlaenge: 38
Maximale Laenge einer Tagestour: 40

```

Das Programm terminiert nach ca. 0.004s.

```

1 ./aufgabe1 muellabfuhr3.txt 2
3 Tag 1: 0 -> 11 -> 1 -> 14 -> 0 -> 13 -> 12 -> 14 -> 0 -> 13 -> 2 -> 12 -> 0 -> 12 ->
5 5 -> 13 -> 0 -> 6 -> 4 -> 5 -> 0 -> 7 -> 4 -> 8 -> 0 -> 10 -> 9 -> 10 -> 0 ->
5 14 -> 13 -> 14 -> 0

```

```

Gesamtlaenge: 32
7 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 ->
9 6 -> 8 -> 0
Gesamtlaenge: 32
11 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 ->
13 9 -> 13 -> 0
Gesamtlaenge: 32
15 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 ->
17 10 -> 11 -> 0 -> 13 -> 6 -> 14 -> 0
Gesamtlaenge: 35
19 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 ->
21 4 -> 14 -> 0 -> 8 -> 2 -> 7 -> 0 -> 7 -> 5 -> 8 -> 0 -> 12 -> 3 -> 11 -> 0 ->
13 13 -> 8 -> 14 -> 0 -> 14 -> 3 -> 13 -> 0 -> 9 -> 3 -> 10 -> 0 -> 11 -> 5 ->
23 14 -> 0 -> 13 -> 10 -> 14 -> 0
Gesamtlaenge: 64
25 Maximale Laenge einer Tagestour: 64

```

5.5 Beispiel 4

Das Programm terminiert nach ca. 0.04s.

```

1 ./aufgabe1 muellabfuhr4.txt 1

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

```

Das Programm terminiert nach ca. 0.003s.

```

./aufgabe1 muellabfuhr4.txt 2

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

```

5.6 Beispiel 5

Das Programm terminiert nach ca. 0.5s.

```

1 ./aufgabe1 muellabfuhr5.txt 1

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

```

```

5 Gesamtlaenge: 1742
6 Tag 2: [...]
7 Gesamtlaenge: 1798
8 Tag 3: [...]
9 Gesamtlaenge: 1762
10 Tag 4: [...]
11 Gesamtlaenge: 1777
12 Tag 5: [...]
13 Gesamtlaenge: 1758
Maximale Laenge einer Tagestour: 1798

```

Das Programm terminiert nach ca. 0.007s.

```

./aufgabe1 muellabfuhr5.txt 2
1
2 Tag 1: [...]
3 Gesamtlaenge: 1502
4 Tag 2: [...]
5 Gesamtlaenge: 1680
6 Tag 3: [...]
7 Gesamtlaenge: 1860
8 Tag 4: [...]
9 Gesamtlaenge: 2071
10 Tag 5: [...]
11 Gesamtlaenge: 2490
Maximale Laenge einer Tagestour: 2490

```

5.7 Beispiel 6

Das Programm terminiert nach ca. 0.17s.

```

./aufgabe1 muellabfuhr6.txt 1
1
2 Durchgefuehrte Iterationen: 34779
3 Tag 1: [...]
4 Gesamtlaenge: 934913
5 Tag 2: [...]
6 Gesamtlaenge: 652524
7 Tag 3: [...]
8 Gesamtlaenge: 841135
9 Tag 4: [...]
10 Gesamtlaenge: 909217
11 Tag 5: [...]
12 Gesamtlaenge: 479374
13 Maximale Laenge einer Tagestour: 934913

```

Das Programm terminiert nach ca. 0.004s.

```

2 ./aufgabe1 muellabfuhr6.txt 2
4 Tag 1: [...]
5 Gesamtlaenge: 1505895
6 Tag 2: [...]
7 Gesamtlaenge: 1597911
8 Tag 3: [...]
9 Gesamtlaenge: 1753919
10 Tag 4: [...]
11 Gesamtlaenge: 2108521
12 Tag 5: [...]
13 Gesamtlaenge: 2593283
14 Maximale Laenge einer Tagestour: 2593283

```

5.8 Beispiel 7

Das Programm terminiert nach ca. 0.7s.

```

2 ./aufgabe1 muellabfuhr7.txt 1

4 Durchgefuehrte Iterationen: 14974
5 Tag 1: [...]
6 Gesamtlaenge: 1470383
7 Tag 2: [...]
8 Gesamtlaenge: 1162790
9 Tag 3: [...]
10 Gesamtlaenge: 1233637
11 Tag 4: [...]
12 Gesamtlaenge: 895341
13 Tag 5: [...]
14 Gesamtlaenge: 839122
Maximale Laenge einer Tagestour: 1470383

```

Das Programm terminiert nach ca. 0.012s.

```

1 ./aufgabe1 muellabfuhr7.txt 2

3 Tag 1: [...]
4 Gesamtlaenge: 33900780
5 Tag 2: [...]
6 Gesamtlaenge: 38035399
7 Tag 3: [...]
8 Gesamtlaenge: 41814152
9 Tag 4: [...]
10 Gesamtlaenge: 46692723
11 Tag 5: [...]
12 Gesamtlaenge: 58776235
13 Maximale Laenge einer Tagestour: 58776235

```

5.9 Beispiel 8

Das Programm terminiert nach ca. 0.8s.

```

./aufgabe1 muellabfuhr8.txt 1

2 Durchgefuehrte Iterationen: 5157
3 Tag 1: [...]
4 Gesamtlaenge: 4956463
5 Tag 2: [...]
6 Gesamtlaenge: 4578770
7 Tag 3: [...]
8 Gesamtlaenge: 4746011
9 Tag 4: [...]
10 Gesamtlaenge: 4236841
11 Tag 5: [...]
12 Gesamtlaenge: 4687578
13 Maximale Laenge einer Tagestour: 4956463

```

Das Programm terminiert nach ca. 0.035s.

```

2 ./aufgabe1 muellabfuhr8.txt 2

4 Tag 1: [...]
5 Gesamtlaenge: 26994078
6 Tag 2: [...]
7 Gesamtlaenge: 30549478
8 Tag 3: [...]

```

```

Gesamtlaenge: 35828505
10 Tag 4: [...]
Gesamtlaenge: 41242037
12 Tag 5: [...]
Gesamtlaenge: 48178252
14 Maximale Laenge einer Tagestour: 48178252

```

6 Quellcode

6.1 Algorithmus 1

6.1.1 Generierung der Ausgangslösung

```

/*
2  * Generiert eine Anfangsloesung fuer den Hill-Climbing Algorithmus.
3  * Kanten, die nahe beieinander liegen werden eher demselben Tag zugeordnet
4  * @param kanten Liste aller Kanten des Graphen
5  * @return Gibt die Anfangsloesung als loesung zurueck
6 */
loesung HillClimbing::genAnfangsLoesung(
7     const std::vector<std::tuple<int, int, int>> &kanten) {

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

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

18        adjazenzListe[a].emplace_back(tup);
19        adjazenzListe[b].emplace_back(tup);
20        kantenIndizes[std::min(a, b)][std::max(a, b)] = index;
21        kantenSpeicher[index - n] = std::make_tuple(a, b, l);
22        index++;
23    }

24    loesung ls;
25    ls.resize(5);

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

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

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

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

36    while (!s.empty()) {

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

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

44            // ind geht von n bis m + n - 1, daher muss n abgezogen werden
45            if (zugewiesen[ind - n]) continue; //Kante wurde bereits einem Tag zugewiesen
46        }
47    }
48}

```

```

58     zugewiesen[ind - n] = true;
59
60     if (anzahlZugewiesen >= proTag && tagIndex != 4) {
61         // Gehe zu naechstem Tag
62         anzahlZugewiesen = 0;
63         tagIndex++;
64     }
65
66     // Speichere Kante
67     ls[tagIndex].emplace_back(ind);
68     anzahlZugewiesen++;
69
70     // Fuege Knoten Stack hinzu, wenn nicht schon geschehen
71     if (!onStack[a]) {
72         s.push(a);
73         onStack[a] = true;
74     }
75     if (!onStack[b]) {
76         s.push(b);
77         onStack[b] = true;
78     }
79 }
80
81 return ls;
}

```

6.1.2 Hill Climbing Algorithmus

```

1  typedef std::vector<std::vector<int>> loesung;
3 #define ZENTRALE 0
4
5 /**
 * Fuehrt den zweiten Algorithmus aus
6 * @param kanten Eine Liste aller Kanten des Graphen
7 */
9 void HillClimbing::run(std::vector<std::tuple<int, int, int>> &kanten) {
10
11     loesung ls = genAnfangsLoesung(kanten);
12
13     const long double alpha = 0.00046;
14     const long long iterationen = 40000 * std::exp(-(n + m) * alpha);
15     long long besterWert = berechneWert(ls);
16
17     for (int i = 0; i < iterationen; ++i) {
18         loesung neueLoesung = berechneNachbar(ls);
19
20         // Berechne Wert der neuen Loesung
21         long long neuerWert = berechneWert(neueLoesung);
22
23         if (neuerWert < besterWert) {
24             // Aktualisiere beste Loesung
25             besterWert = neuerWert;
26             ls = neueLoesung;
27         }
28     }
29
30     ls = entferneMehrfachAbgefahreneStrassen(ls);
31
32     // Berechne Laenge fuer alle Tagesstrecken
33     std::vector<long long> laenge(5);
34     long long wert = 0;
35     for (int i = 0; i < 5; ++i) {
36         laenge[i] = berechneTagesstrecke(ls[i]);
37         wert = std::max(wert, laenge[i]);
38
39         // Konvertiere Loesung in richtige Pfade im Graphen
40         ls[i] = konvertiereZuNormal(ls[i]);
41     }
42
43 }

```

```

41     }
43     // IO
44     // [...]
45 }

47 /**
 * Berechnet einen Nachbarn der gegebenen Loesung im Suchraum.
49 * @param s Eine Loesung
50 * @return Gibt die gegebene Loesung leicht veraendert zurueck
51 */
52 loesung HillClimbing::berechneNachbar(loesung s) {
53
54     // Bestimme Tag mit laengster Strecke
55     int maxIndex = 0;
56     long long maxWert = 0;
57     for (int i = 0; i < 5; ++i) {
58         long long val = berechneTagesstrecke(s[i]);
59         if (val > maxWert) {
60             maxWert = val;
61             maxIndex = i;
62         }
63     }
64
65     // Tausche Kante zwischen zwei Tagen
66     int tag1 = maxIndex;
67     // Bestimme zufaellig einen anderen Tag zum Tauschen
68     unsigned int tag2 = tag1;
69     while (tag2 == tag1) {
70         tag2 = zufaelligeZahl(0, 4);
71     }
72
73     if (!s[tag1].empty()) {
74         unsigned int index = zufaelligeZahl(0, s[tag1].size() - 1);
75         int strasse = s[tag1][index];
76         s[tag1].erase(s[tag1].begin() + index);
77         s[tag2].emplace_back(strasse);
78     }
79
80     // Tausche Reihenfolge der Strassen an einem Tag
81     if (s[tag1].size() >= 2) {
82         unsigned int index1 = zufaelligeZahl(0, s[tag1].size() - 1);
83         unsigned int index2 = zufaelligeZahl(0, s[tag1].size() - 1);
84
85         std::swap(s[tag1][index1], s[tag1][index2]);
86     }
87
88     return s;
89 }

90 /**
 * Berechnet den Wert einer Loesung. Gibt die Laenge der laengsten Tagesstrecke
91 * zurueck
 * @param s Eine Loesung
92 */
93 long long HillClimbing::berechneWert(const loesung &s) { // O(m)
94     long long maxTotal = 0;
95
96     for (const auto &pfad: s) {
97         maxTotal = std::max(maxTotal, berechneTagesstrecke(pfad));
98     }
99
100    return maxTotal;
101 }

102 /**
103 * Berechnet die Laenge einer Tagesstrecke
104 * @param tagesstrecke Eine Liste an Kanten
105 * @return Gibt die Laenge des kuerzesten Pfades zurueck, der die Kanten in der gegebenen
106 * Reihenfolge abfaehrt
107 */
108 long long HillClimbing::berechneTagesstrecke(const std::vector<int> &tagesstrecke) {
109     int vorherigerKnoten = ZENTRALE;
110
111     for (int i = 1; i < tagesstrecke.size(); ++i) {
112         int aktuellerKnoten = tagesstrecke[i];
113
114         if (aktuellerKnoten == vorherigerKnoten) {
115             continue;
116         }
117
118         if (aktuellerKnoten == 0) {
119             vorherigerKnoten = aktuellerKnoten;
120             continue;
121         }
122
123         if (aktuellerKnoten == 1) {
124             vorherigerKnoten = aktuellerKnoten;
125             continue;
126         }
127
128         if (aktuellerKnoten == 2) {
129             vorherigerKnoten = aktuellerKnoten;
130             continue;
131         }
132
133         if (aktuellerKnoten == 3) {
134             vorherigerKnoten = aktuellerKnoten;
135             continue;
136         }
137
138         if (aktuellerKnoten == 4) {
139             vorherigerKnoten = aktuellerKnoten;
140             continue;
141         }
142
143         if (aktuellerKnoten == 5) {
144             vorherigerKnoten = aktuellerKnoten;
145             continue;
146         }
147
148         if (aktuellerKnoten == 6) {
149             vorherigerKnoten = aktuellerKnoten;
150             continue;
151         }
152
153         if (aktuellerKnoten == 7) {
154             vorherigerKnoten = aktuellerKnoten;
155             continue;
156         }
157
158         if (aktuellerKnoten == 8) {
159             vorherigerKnoten = aktuellerKnoten;
160             continue;
161         }
162
163         if (aktuellerKnoten == 9) {
164             vorherigerKnoten = aktuellerKnoten;
165             continue;
166         }
167
168         if (aktuellerKnoten == 10) {
169             vorherigerKnoten = aktuellerKnoten;
170             continue;
171         }
172
173         if (aktuellerKnoten == 11) {
174             vorherigerKnoten = aktuellerKnoten;
175             continue;
176         }
177
178         if (aktuellerKnoten == 12) {
179             vorherigerKnoten = aktuellerKnoten;
180             continue;
181         }
182
183         if (aktuellerKnoten == 13) {
184             vorherigerKnoten = aktuellerKnoten;
185             continue;
186         }
187
188         if (aktuellerKnoten == 14) {
189             vorherigerKnoten = aktuellerKnoten;
190             continue;
191         }
192
193         if (aktuellerKnoten == 15) {
194             vorherigerKnoten = aktuellerKnoten;
195             continue;
196         }
197
198         if (aktuellerKnoten == 16) {
199             vorherigerKnoten = aktuellerKnoten;
200             continue;
201         }
202
203         if (aktuellerKnoten == 17) {
204             vorherigerKnoten = aktuellerKnoten;
205             continue;
206         }
207
208         if (aktuellerKnoten == 18) {
209             vorherigerKnoten = aktuellerKnoten;
210             continue;
211         }
212
213         if (aktuellerKnoten == 19) {
214             vorherigerKnoten = aktuellerKnoten;
215             continue;
216         }
217
218         if (aktuellerKnoten == 20) {
219             vorherigerKnoten = aktuellerKnoten;
220             continue;
221         }
222
223         if (aktuellerKnoten == 21) {
224             vorherigerKnoten = aktuellerKnoten;
225             continue;
226         }
227
228         if (aktuellerKnoten == 22) {
229             vorherigerKnoten = aktuellerKnoten;
230             continue;
231         }
232
233         if (aktuellerKnoten == 23) {
234             vorherigerKnoten = aktuellerKnoten;
235             continue;
236         }
237
238         if (aktuellerKnoten == 24) {
239             vorherigerKnoten = aktuellerKnoten;
240             continue;
241         }
242
243         if (aktuellerKnoten == 25) {
244             vorherigerKnoten = aktuellerKnoten;
245             continue;
246         }
247
248         if (aktuellerKnoten == 26) {
249             vorherigerKnoten = aktuellerKnoten;
250             continue;
251         }
252
253         if (aktuellerKnoten == 27) {
254             vorherigerKnoten = aktuellerKnoten;
255             continue;
256         }
257
258         if (aktuellerKnoten == 28) {
259             vorherigerKnoten = aktuellerKnoten;
260             continue;
261         }
262
263         if (aktuellerKnoten == 29) {
264             vorherigerKnoten = aktuellerKnoten;
265             continue;
266         }
267
268         if (aktuellerKnoten == 30) {
269             vorherigerKnoten = aktuellerKnoten;
270             continue;
271         }
272
273         if (aktuellerKnoten == 31) {
274             vorherigerKnoten = aktuellerKnoten;
275             continue;
276         }
277
278         if (aktuellerKnoten == 32) {
279             vorherigerKnoten = aktuellerKnoten;
280             continue;
281         }
282
283         if (aktuellerKnoten == 33) {
284             vorherigerKnoten = aktuellerKnoten;
285             continue;
286         }
287
288         if (aktuellerKnoten == 34) {
289             vorherigerKnoten = aktuellerKnoten;
290             continue;
291         }
292
293         if (aktuellerKnoten == 35) {
294             vorherigerKnoten = aktuellerKnoten;
295             continue;
296         }
297
298         if (aktuellerKnoten == 36) {
299             vorherigerKnoten = aktuellerKnoten;
300             continue;
301         }
302
303         if (aktuellerKnoten == 37) {
304             vorherigerKnoten = aktuellerKnoten;
305             continue;
306         }
307
308         if (aktuellerKnoten == 38) {
309             vorherigerKnoten = aktuellerKnoten;
310             continue;
311         }
312
313         if (aktuellerKnoten == 39) {
314             vorherigerKnoten = aktuellerKnoten;
315             continue;
316         }
317
318         if (aktuellerKnoten == 40) {
319             vorherigerKnoten = aktuellerKnoten;
320             continue;
321         }
322
323         if (aktuellerKnoten == 41) {
324             vorherigerKnoten = aktuellerKnoten;
325             continue;
326         }
327
328         if (aktuellerKnoten == 42) {
329             vorherigerKnoten = aktuellerKnoten;
330             continue;
331         }
332
333         if (aktuellerKnoten == 43) {
334             vorherigerKnoten = aktuellerKnoten;
335             continue;
336         }
337
338         if (aktuellerKnoten == 44) {
339             vorherigerKnoten = aktuellerKnoten;
340             continue;
341         }
342
343         if (aktuellerKnoten == 45) {
344             vorherigerKnoten = aktuellerKnoten;
345             continue;
346         }
347
348         if (aktuellerKnoten == 46) {
349             vorherigerKnoten = aktuellerKnoten;
350             continue;
351         }
352
353         if (aktuellerKnoten == 47) {
354             vorherigerKnoten = aktuellerKnoten;
355             continue;
356         }
357
358         if (aktuellerKnoten == 48) {
359             vorherigerKnoten = aktuellerKnoten;
360             continue;
361         }
362
363         if (aktuellerKnoten == 49) {
364             vorherigerKnoten = aktuellerKnoten;
365             continue;
366         }
367
368         if (aktuellerKnoten == 50) {
369             vorherigerKnoten = aktuellerKnoten;
370             continue;
371         }
372
373         if (aktuellerKnoten == 51) {
374             vorherigerKnoten = aktuellerKnoten;
375             continue;
376         }
377
378         if (aktuellerKnoten == 52) {
379             vorherigerKnoten = aktuellerKnoten;
380             continue;
381         }
382
383         if (aktuellerKnoten == 53) {
384             vorherigerKnoten = aktuellerKnoten;
385             continue;
386         }
387
388         if (aktuellerKnoten == 54) {
389             vorherigerKnoten = aktuellerKnoten;
390             continue;
391         }
392
393         if (aktuellerKnoten == 55) {
394             vorherigerKnoten = aktuellerKnoten;
395             continue;
396         }
397
398         if (aktuellerKnoten == 56) {
399             vorherigerKnoten = aktuellerKnoten;
400             continue;
401         }
402
403         if (aktuellerKnoten == 57) {
404             vorherigerKnoten = aktuellerKnoten;
405             continue;
406         }
407
408         if (aktuellerKnoten == 58) {
409             vorherigerKnoten = aktuellerKnoten;
410             continue;
411         }
412
413         if (aktuellerKnoten == 59) {
414             vorherigerKnoten = aktuellerKnoten;
415             continue;
416         }
417
418         if (aktuellerKnoten == 60) {
419             vorherigerKnoten = aktuellerKnoten;
420             continue;
421         }
422
423         if (aktuellerKnoten == 61) {
424             vorherigerKnoten = aktuellerKnoten;
425             continue;
426         }
427
428         if (aktuellerKnoten == 62) {
429             vorherigerKnoten = aktuellerKnoten;
430             continue;
431         }
432
433         if (aktuellerKnoten == 63) {
434             vorherigerKnoten = aktuellerKnoten;
435             continue;
436         }
437
438         if (aktuellerKnoten == 64) {
439             vorherigerKnoten = aktuellerKnoten;
440             continue;
441         }
442
443         if (aktuellerKnoten == 65) {
444             vorherigerKnoten = aktuellerKnoten;
445             continue;
446         }
447
448         if (aktuellerKnoten == 66) {
449             vorherigerKnoten = aktuellerKnoten;
450             continue;
451         }
452
453         if (aktuellerKnoten == 67) {
454             vorherigerKnoten = aktuellerKnoten;
455             continue;
456         }
457
458         if (aktuellerKnoten == 68) {
459             vorherigerKnoten = aktuellerKnoten;
460             continue;
461         }
462
463         if (aktuellerKnoten == 69) {
464             vorherigerKnoten = aktuellerKnoten;
465             continue;
466         }
467
468         if (aktuellerKnoten == 70) {
469             vorherigerKnoten = aktuellerKnoten;
470             continue;
471         }
472
473         if (aktuellerKnoten == 71) {
474             vorherigerKnoten = aktuellerKnoten;
475             continue;
476         }
477
478         if (aktuellerKnoten == 72) {
479             vorherigerKnoten = aktuellerKnoten;
480             continue;
481         }
482
483         if (aktuellerKnoten == 73) {
484             vorherigerKnoten = aktuellerKnoten;
485             continue;
486         }
487
488         if (aktuellerKnoten == 74) {
489             vorherigerKnoten = aktuellerKnoten;
490             continue;
491         }
492
493         if (aktuellerKnoten == 75) {
494             vorherigerKnoten = aktuellerKnoten;
495             continue;
496         }
497
498         if (aktuellerKnoten == 76) {
499             vorherigerKnoten = aktuellerKnoten;
500             continue;
501         }
502
503         if (aktuellerKnoten == 77) {
504             vorherigerKnoten = aktuellerKnoten;
505             continue;
506         }
507
508         if (aktuellerKnoten == 78) {
509             vorherigerKnoten = aktuellerKnoten;
510             continue;
511         }
512
513         if (aktuellerKnoten == 79) {
514             vorherigerKnoten = aktuellerKnoten;
515             continue;
516         }
517
518         if (aktuellerKnoten == 80) {
519             vorherigerKnoten = aktuellerKnoten;
520             continue;
521         }
522
523         if (aktuellerKnoten == 81) {
524             vorherigerKnoten = aktuellerKnoten;
525             continue;
526         }
527
528         if (aktuellerKnoten == 82) {
529             vorherigerKnoten = aktuellerKnoten;
530             continue;
531         }
532
533         if (aktuellerKnoten == 83) {
534             vorherigerKnoten = aktuellerKnoten;
535             continue;
536         }
537
538         if (aktuellerKnoten == 84) {
539             vorherigerKnoten = aktuellerKnoten;
540             continue;
541         }
542
543         if (aktuellerKnoten == 85) {
544             vorherigerKnoten = aktuellerKnoten;
545             continue;
546         }
547
548         if (aktuellerKnoten == 86) {
549             vorherigerKnoten = aktuellerKnoten;
550             continue;
551         }
552
553         if (aktuellerKnoten == 87) {
554             vorherigerKnoten = aktuellerKnoten;
555             continue;
556         }
557
558         if (aktuellerKnoten == 88) {
559             vorherigerKnoten = aktuellerKnoten;
560             continue;
561         }
562
563         if (aktuellerKnoten == 89) {
564             vorherigerKnoten = aktuellerKnoten;
565             continue;
566         }
567
568         if (aktuellerKnoten == 90) {
569             vorherigerKnoten = aktuellerKnoten;
570             continue;
571         }
572
573         if (aktuellerKnoten == 91) {
574             vorherigerKnoten = aktuellerKnoten;
575             continue;
576         }
577
578         if (aktuellerKnoten == 92) {
579             vorherigerKnoten = aktuellerKnoten;
580             continue;
581         }
582
583         if (aktuellerKnoten == 93) {
584             vorherigerKnoten = aktuellerKnoten;
585             continue;
586         }
587
588         if (aktuellerKnoten == 94) {
589             vorherigerKnoten = aktuellerKnoten;
590             continue;
591         }
592
593         if (aktuellerKnoten == 95) {
594             vorherigerKnoten = aktuellerKnoten;
595             continue;
596         }
597
598         if (aktuellerKnoten == 96) {
599             vorherigerKnoten = aktuellerKnoten;
600             continue;
601         }
602
603         if (aktuellerKnoten == 97) {
604             vorherigerKnoten = aktuellerKnoten;
605             continue;
606         }
607
608         if (aktuellerKnoten == 98) {
609             vorherigerKnoten = aktuellerKnoten;
610             continue;
611         }
612
613         if (aktuellerKnoten == 99) {
614             vorherigerKnoten = aktuellerKnoten;
615             continue;
616         }
617
618         if (aktuellerKnoten == 100) {
619             vorherigerKnoten = aktuellerKnoten;
620             continue;
621         }
622
623         if (aktuellerKnoten == 101) {
624             vorherigerKnoten = aktuellerKnoten;
625             continue;
626         }
627
628         if (aktuellerKnoten == 102) {
629             vorherigerKnoten = aktuellerKnoten;
630             continue;
631         }
632
633         if (aktuellerKnoten == 103) {
634             vorherigerKnoten = aktuellerKnoten;
635             continue;
636         }
637
638         if (aktuellerKnoten == 104) {
639             vorherigerKnoten = aktuellerKnoten;
640             continue;
641         }
642
643         if (aktuellerKnoten == 105) {
644             vorherigerKnoten = aktuellerKnoten;
645             continue;
646         }
647
648         if (aktuellerKnoten == 106) {
649             vorherigerKnoten = aktuellerKnoten;
650             continue;
651         }
652
653         if (aktuellerKnoten == 107) {
654             vorherigerKnoten = aktuellerKnoten;
655             continue;
656         }
657
658         if (aktuellerKnoten == 108) {
659             vorherigerKnoten = aktuellerKnoten;
660             continue;
661         }
662
663         if (aktuellerKnoten == 109) {
664             vorherigerKnoten = aktuellerKnoten;
665             continue;
666         }
667
668         if (aktuellerKnoten == 110) {
669             vorherigerKnoten = aktuellerKnoten;
670             continue;
671         }
672
673         if (aktuellerKnoten == 111) {
674             vorherigerKnoten = aktuellerKnoten;
675             continue;
676         }
677
678         if (aktuellerKnoten == 112) {
679             vorherigerKnoten = aktuellerKnoten;
680             continue;
681         }
682
683         if (aktuellerKnoten == 113) {
684             vorherigerKnoten = aktuellerKnoten;
685             continue;
686         }
687
688         if (aktuellerKnoten == 114) {
689             vorherigerKnoten = aktuellerKnoten;
690             continue;
691         }
692
693         if (aktuellerKnoten == 115) {
694             vorherigerKnoten = aktuellerKnoten;
695             continue;
696         }
697
698         if (aktuellerKnoten == 116) {
699             vorherigerKnoten = aktuellerKnoten;
700             continue;
701         }
702
703         if (aktuellerKnoten == 117) {
704             vorherigerKnoten = aktuellerKnoten;
705             continue;
706         }
707
708         if (aktuellerKnoten == 118) {
709             vorherigerKnoten = aktuellerKnoten;
710             continue;
711         }
712
713         if (aktuellerKnoten == 119) {
714             vorherigerKnoten = aktuellerKnoten;
715             continue;
716         }
717
718         if (aktuellerKnoten == 120) {
719             vorherigerKnoten = aktuellerKnoten;
720             continue;
721         }
722
723         if (aktuellerKnoten == 121) {
724             vorherigerKnoten = aktuellerKnoten;
725             continue;
726         }
727
728         if (aktuellerKnoten == 122) {
729             vorherigerKnoten = aktuellerKnoten;
730             continue;
731         }
732
733         if (aktuellerKnoten == 123) {
734             vorherigerKnoten = aktuellerKnoten;
735             continue;
736         }
737
738         if (aktuellerKnoten == 124) {
739             vorherigerKnoten = aktuellerKnoten;
740             continue;
741         }
742
743         if (aktuellerKnoten == 125) {
744             vorherigerKnoten = aktuellerKnoten;
745             continue;
746         }
747
748         if (aktuellerKnoten == 126) {
749             vorherigerKnoten = aktuellerKnoten;
750             continue;
751         }
752
753         if (aktuellerKnoten == 127) {
754             vorherigerKnoten = aktuellerKnoten;
755             continue;
756         }
757
758         if (aktuellerKnoten == 128) {
759             vorherigerKnoten = aktuellerKnoten;
760             continue;
761         }
762
763         if (aktuellerKnoten == 129) {
764             vorherigerKnoten = aktuellerKnoten;
765             continue;
766         }
767
768         if (aktuellerKnoten == 130) {
769             vorherigerKnoten = aktuellerKnoten;
770             continue;
771         }
772
773         if (aktuellerKnoten == 131) {
774             vorherigerKnoten = aktuellerKnoten;
775             continue;
776         }
777
778         if (aktuellerKnoten == 132) {
779             vorherigerKnoten = aktuellerKnoten;
780             continue;
781         }
782
783         if (aktuellerKnoten == 133) {
784             vorherigerKnoten = aktuellerKnoten;
785             continue;
786         }
787
788         if (aktuellerKnoten == 134) {
789             vorherigerKnoten = aktuellerKnoten;
790             continue;
791         }
792
793         if (aktuellerKnoten == 135) {
794             vorherigerKnoten = aktuellerKnoten;
795             continue;
796         }
797
798         if (aktuellerKnoten == 136) {
799             vorherigerKnoten = aktuellerKnoten;
800             continue;
801         }
802
803         if (aktuellerKnoten == 137) {
804             vorherigerKnoten = aktuellerKnoten;
805             continue;
806         }
807
808         if (aktuellerKnoten == 138) {
809             vorherigerKnoten = aktuellerKnoten;
810             continue;
811         }
812
813         if (aktuellerKnoten == 139) {
814             vorherigerKnoten = aktuellerKnoten;
815             continue;
816         }
817
818         if (aktuellerKnoten == 140) {
819             vorherigerKnoten = aktuellerKnoten;
820             continue;
821         }
822
823         if (aktuellerKnoten == 141) {
824             vorherigerKnoten = aktuellerKnoten;
825             continue;
826         }
827
828         if (aktuellerKnoten == 142) {
829             vorherigerKnoten = aktuellerKnoten;
830             continue;
831         }
832
833         if (aktuellerKnoten == 143) {
834             vorherigerKnoten = aktuellerKnoten;
835             continue;
836         }
837
838         if (aktuellerKnoten == 144) {
839             vorherigerKnoten = aktuellerKnoten;
840             continue;
841         }
842
843         if (aktuellerKnoten == 145) {
844             vorherigerKnoten = aktuellerKnoten;
845             continue;
846         }
847
848         if (aktuellerKnoten == 146) {
849             vorherigerKnoten = aktuellerKnoten;
850             continue;
851         }
852
853         if (aktuellerKnoten == 147) {
854             vorherigerKnoten = aktuellerKnoten;
855             continue;
856         }
857
858         if (aktuellerKnoten == 148) {
859             vorherigerKnoten = aktuellerKnoten;
860             continue;
861         }
862
863         if (aktuellerKnoten == 149) {
864             vorherigerKnoten = aktuellerKnoten;
865             continue;
866         }
867
868         if (aktuellerKnoten == 150) {
869             vorherigerKnoten = aktuellerKnoten;
870             continue;
871         }
872
873         if (aktuellerKnoten == 151) {
874             vorherigerKnoten = aktuellerKnoten;
875             continue;
876         }
877
878         if (aktuellerKnoten == 152) {
879             vorherigerKnoten = aktuellerKnoten;
880             continue;
881         }
882
883         if (aktuellerKnoten == 153) {
884             vorherigerKnoten = aktuellerKnoten;
885             continue;
886         }
887
888         if (aktuellerKnoten == 154) {
889             vorherigerKnoten = aktuellerKnoten;
890             continue;
891         }
892
893         if (aktuellerKnoten == 155) {
894             vorherigerKnoten = aktuellerKnoten;
895             continue;
896         }
897
898         if (aktuellerKnoten == 156) {
899             vorherigerKnoten = aktuellerKnoten;
900             continue;
901         }
902
903         if (aktuellerKnoten == 157) {
904             vorherigerKnoten = aktuellerKnoten;
905             continue;
906         }
907
908         if (aktuellerKnoten == 158) {
909             vorherigerKnoten = aktuellerKnoten;
910             continue;
911         }
912
913         if (aktuellerKnoten == 159) {
914             vorherigerKnoten = aktuellerKnoten;
915             continue;
916         }
917
918         if (aktuellerKnoten == 160) {
919             vorherigerKnoten = aktuellerKnoten;
920             continue;
921         }
922
923         if (aktuellerKnoten == 161) {
924             vorherigerKnoten = aktuellerKnoten;
925             continue;
926         }
927
928         if (aktuellerKnoten == 162) {
929             vorherigerKnoten = aktuellerKnoten;
930             continue;
931         }
932
933         if (aktuellerKnoten == 163) {
934             vorherigerKnoten = aktuellerKnoten;
935             continue;
936         }
937
938         if (aktuellerKnoten == 164) {
939             vorherigerKnoten = aktuellerKnoten;
940             continue;
941         }
942
943         if (aktuellerKnoten == 165) {
944             vorherigerKnoten = aktuellerKnoten;
945             continue;
946         }
947
948         if (aktuellerKnoten == 166) {
949             vorherigerKnoten = aktuellerKnoten;
950             continue;
951         }
952
953         if (aktuellerKnoten == 167) {
954             vorherigerKnoten = aktuellerKnoten;
955             continue;
956         }
957
958         if (aktuellerKnoten == 168) {
959             vorherigerKnoten = aktuellerKnoten;
960             continue;
961         }
962
963         if (aktuellerKnoten == 169) {
964             vorherigerKnoten = aktuellerKnoten;
965             continue;
966         }
967
968         if (aktuellerKnoten == 170) {
969             vorherigerKnoten = aktuellerKnoten;
970             continue;
971         }
972
973         if (aktuellerKnoten == 171) {
974             vorherigerKnoten = aktuellerKnoten;
975             continue;
976         }
977
978         if (aktuellerKnoten == 172) {
979             vorherigerKnoten = aktuellerKnoten;
980             continue;
981         }
982
983         if (aktuellerKnoten == 173) {
984             vorherigerKnoten = aktuellerKnoten;
985             continue;
986         }
987
988         if (aktuellerKnoten == 174) {
989             vorherigerKnoten = aktuellerKnoten;
990             continue;
991         }
992
993         if (aktuellerKnoten == 175) {
994             vorherigerKnoten = aktuellerKnoten;
995             continue;
996         }
997
998         if (aktuellerKnoten == 176) {
999             vorherigerKnoten = aktuellerKnoten;
1000            continue;
1001        }
1002
1003         if (aktuellerKnoten == 100) {
1004             vorherigerKnoten = aktuellerKnoten;
1005            continue;
1006        }
1007
1008         if (aktuellerKnoten == 101) {
1009             vorherigerKnoten = aktuellerKnoten;
1010            continue;
1011        }
1012
1013         if (aktuellerKnoten == 102) {
1014             vorherigerKnoten = aktuellerKnoten;
1015            continue;
1016        }
1017
1018         if (aktuellerKnoten == 103) {
1019             vorherigerKnoten = aktuellerKnoten;
1020            continue;
1021        }
1022
1023         if (aktuellerKnoten == 104) {
1024             vorherigerKnoten = aktuellerKnoten;
1025            continue;
1026        }
1027
1028         if (aktuellerKnoten == 105) {
1029             vorherigerKnoten = aktuellerKnoten;
1030            continue;
1031        }
1032
1033         if (aktuellerKnoten == 106) {
1034             vorherigerKnoten = aktuellerKnoten;
1035            continue;
1036        }
1037
1038         if (aktuellerKnoten == 107) {
1039             vorherigerKnoten = aktuellerKnoten;
1040            continue;
1041        }
1042
1043         if (aktuellerKnoten == 108) {
1044             vorherigerKnoten = aktuellerKnoten;
1045            continue;
1046        }
1047
1048         if (aktuellerKnoten == 109) {
1049             vorherigerKnoten = aktuellerKnoten;
1050            continue;
1051        }
1052
1053         if (aktuellerKnoten == 110) {
1054             vorherigerKnoten = aktuellerKnoten;
1055            continue;
1056        }
1057
1058         if (aktuellerKnoten == 111) {
1059             vorherigerKnoten = aktuellerKnoten;
1060            continue;
1061        }
1062
1063         if (aktuellerKnoten == 112) {
1064             vorherigerKnoten = aktuellerKnoten;
1065            continue;
1066        }
1067
1068         if (aktuellerKnoten == 113) {
1069             vorherigerKnoten = aktuellerKnoten;
1070            continue;
1071        }
1072
1073         if (aktuellerKnoten == 114) {
1074             vorherigerKnoten = aktuellerKnoten;
1075            continue;
1076        }
1077
1078         if (aktuellerKnoten == 115) {
1079             vorherigerKnoten = aktuellerKnoten;
1080            continue;
1081        }
1082
1083         if (aktuellerKnoten == 116) {
1084             vorherigerKnoten = aktuellerKnoten;
1085            continue;
1086        }
1087
1088         if (aktuellerKnoten == 117) {
1089             vorherigerKnoten = aktuellerKnoten;
1090            continue;
1091        }
1092
1093         if (aktuellerKnoten == 118) {
1094             vorherigerKnoten = aktuellerKnoten;
1095            continue;
1096        }
1097
1098         if (aktuellerKnoten == 119) {
1099             vorherigerKnoten = aktuellerKnoten;
1100            continue;
1101        }
1102
1103         if (aktuellerKnoten == 110) {
1104             vorherigerKnoten = aktuellerKnoten;
1105            continue;
1106        }
1107
1108         if (aktuellerKnoten == 111) {
1109             vorherigerKnoten = aktuellerKnoten;
1110            continue;
1111        }
1112
1113         if
```

```

115     long long gesamtStrecke = 0;
116
117     for (int kante: tagesstrecke) {
118         auto r = berechneStrecke(vorherigerKnoten, kante);
119         gesamtStrecke += r.first;
120         vorherigerKnoten = r.second;
121     }
122
123     // Strecke zurueck zur Zentrale addieren
124     gesamtStrecke += berechneStrecke(vorherigerKnoten, ZENTRALE).first;
125
126     return gesamtStrecke;
127 }
128
129 /**
130 * Berechnet die Entfernung zwischen den Knoten s und t
131 * @param s Der Startknoten, muss ein originaler Knoten sein
132 * @param t Der Zielknoten
133 * @return Gibt die Laenge der Strecke zurueck und den originalen Knoten, an dem man sich
134 *         danach befindet.
135 */
136 std::pair<long long, int> HillClimbing::berechneStrecke(int s, int t) {
137     // Stelle sicher, dass s ein echter Knoten ist
138     if (t == s) {
139         return {0, s};
140     }
141
142     int knoten1 = t;
143     int knoten2 = t;
144
145     if (t >= n) {
146         auto k1 = get<0>(kantenSpeicher[t - n]);
147         auto k2 = get<1>(kantenSpeicher[t - n]);
148
149         if (distanz[s][k1] < distanz[s][k2]) {
150             knoten1 = k1;
151             knoten2 = k2;
152         } else {
153             knoten1 = k2;
154             knoten2 = k1;
155         }
156     }
157
158     return {distanz[s][knoten1] + adjMatrix[knoten1][knoten2], knoten2};
159 }
```

6.1.3 Versuch mehrfaches Abfahren von Straßen zu vermeiden

```

/***
2  * Versucht zu minimieren, dass Kanten unnoetigerweise mehrfach abgefahren werden
3  * @param s Eine Loesung
4  * @return Gibt die verbesserte Loesung zurueck
*/
5 loesung HillClimbing::entferneMehrfachAbgefaehrteStrassen(loesung s) {
6     std::unordered_map<int, long long> tagesstrecken;
7     tagesstrecken.reserve(10);
8     for (int i = 0; i < 5; ++i) { // 0(m)
9         if (s[i].empty()) continue;
10        // Nutze erste Kante in s[i] als Key, da jede Kante nur einem Tag zugeordnet ist
11        // -> s[i][0] ist eindeutig
12        tagesstrecken[s[i][0]] = berechneTagesstrecke(s[i]);
13    }
14
15    // Sortiere nach aufsteigender Laenge
16    std::sort(s.begin(), s.end(), [&](auto &a, auto &b) {
17        if (a.empty()) {
18            return true;
19        } else if (b.empty()) {
20            return false;
21        }
22    });
23 }
```

```

24     return tagesstrecken[a[0]] < tagesstrecken[b[0]];
25 };
26
27 for (int i = 0; i < 5; ++i) {
28     std::vector<bool> abgefahren(m);
29
30     std::vector<int> normal = konvertiereZuNormal(s[i]);
31     // Speichere welche Kanten in Tagesstrecke i abgefahren werden
32     for (int k = 1; k < normal.size(); ++k) {
33         int a = normal[k - 1];
34         int b = normal[k];
35         int index = kantenIndizes[std::min(a, b)][std::max(a, b)];
36         // Index geht von n bis n + m - 1
37         abgefahren[index - n] = true;
38     }
39
40     // Entferne die gefundenen Kanten aus den anderen Tagesstrecken
41     for (int j = i + 1; j < 5; ++j) {
42         for (int k = 0; k < s[j].size(); ++k) {
43             if (abgefahren[s[j][k] - n]) {
44                 // Markieren fuer spaetere Loeschung
45                 s[j][k] = -1;
46             }
47         }
48         // Loesche zuvor markierte Elemente
49         s[j].erase(std::remove(s[j].begin(), s[j].end(), -1), s[j].end());
50     }
51 }
52 return s;
53
54 /**
55 * Wandelt die gegebene Liste an Kanten in einem richtigen Pfad im Graphen um
56 * @param tagesstrecke Eine Liste an Kanten
57 * @return Gibt eine Liste an zusammenhaengenden Knoten zurueck, die die in tagesstrecke
58 *        gegebenen Kanten enthaelt
59 */
60 std::vector<int>
61 HillClimbing::konvertiereZuNormal(std::vector<int> tagesstrecke) {
62
63     std::vector<int> normalerPfad;
64     int vorherigerKnoten = 0;
65
66     tagesstrecke.emplace_back(ZENTRALE); // Pfad muss bei der Zentrale enden
67     for (int kante: tagesstrecke) {
68         int ziel = kante;
69
70         // Ueberprueft, ob "kante" wirklich eine Kante ist oder ein Knoten
71         if (kante >= n) {
72             // ist eine Kante
73
74             // Die beiden Knoten, die durch die momentane Kante verbunden sind
75             auto knoten1 = get<0>(kantenSpeicher[kante - n]);
76             auto knoten2 = get<1>(kantenSpeicher[kante - n]);
77
78             // Bestimme, welcher der beiden Knoten naeher an momentaner Position ist
79             if (distanz[vorherigerKnoten][knoten1] <
80                 distanz[vorherigerKnoten][knoten2]) {
81                 ziel = knoten1;
82             } else {
83                 ziel = knoten2;
84             }
85         }
86
87         normalerPfad.emplace_back(vorherigerKnoten);
88
89         // Bestimme und speichere Pfad von vorherigerKnoten zu ziel
90         do {
91             vorherigerKnoten = parent[vorherigerKnoten][ziel];
92             if (vorherigerKnoten == normalerPfad[normalerPfad.size() - 1]) {
93                 // Wurde bereits gespeichert -> ueberspringen
94                 break;
95             }
96         }
97     }
98 }
```

```

96         }
97         normalerPfad.emplace_back(vorherigerKnoten);
98     } while (vorherigerKnoten != ziel);

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

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

107        if (ziel == node1) {
108            ziel = node2;
109        } else {
110            ziel = node1;
111        }
112    }

113    vorherigerKnoten = ziel;
114}

115}

116}

117}

118}

119}

120 return normalerPfad;
}

```

6.2 Algorithmus 2

6.2.1 Berechnen der Pfade

```

2 /**
 * Berechnet die kuerzesten Wege zu allen Kanten des originalen Graphen
4 * @return Gibt die Pfade zu den Kanten mit Laenge zurueck
 */
6 std::vector<std::pair<std::vector<int>, long long>> Algorithmus2::berechnePfade() {

8     // parent[v] speichert den naechsten Knoten auf dem kuerzesten Pfad von v zur Zentrale
9     std::vector<std::pair<int, int>> parent(graph.size());
10
11    std::priority_queue<std::pair<int, int>> pq;
12    std::vector<bool> visited(graph.size());

14    // Speichert die Distanz von der Zentrale zu den Knoten
15    std::vector<int> dist(graph.size(), INT32_MAX);
16
17    dist[0] = 0;
18    pq.push(std::make_pair(0, 0));

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

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

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

32            if (dist[knoten] + kantenGewicht < dist[adjKnoten]) {
33                dist[adjKnoten] = dist[knoten] + kantenGewicht;
34                pq.push(std::make_pair(dist[adjKnoten] * (-1), adjKnoten));
35                parent[adjKnoten] = std::make_pair(knoten, kantenGewicht);
36            }
37        }
38    }
}

```

```

40     std::vector<std::pair<std::vector<int>, long long>> pfade;
42     pfade.reserve(m);
43
44     // Liste mit Indizes der Kanten, um sie zu sortieren
45     std::vector<int> kanten(m);
46     for (int i = n; i < n + m; ++i) {
47         kanten[i - n] = i;
48     }
49
50     // Sortiere Liste mit Indizes der Kanten nach aufsteigender Entfernung zur Zentrale
51     std::sort(kanten.begin(), kanten.end(), [&](int a, int b) {
52         return dist[a] > dist[b];
53     });
54
55     // Speichert, welche Kanten bereits auf dem Weg zu anderen Kanten abgefahren wurde
56     // und somit nicht weiter betrachtet werden muessen
57     std::vector<int> bereitsAbgefahren(n + m);
58
59     for (int i: kanten) { // O(m)
60         if (bereitsAbgefahren[i] > 0) continue;
61         int kante = i;
62         std::pair<int, std::pair<int, int>> tmp;
63
64         for (auto adj: graph[kante]) {
65             if (adj.first != parent[kante].first) {
66                 tmp.first = adj.first;
67                 tmp.second = parent[adj.first];
68                 parent[adj.first] = std::make_pair(kante, adj.second);
69                 kante = adj.first;
70                 break;
71             }
72         }
73
74         std::vector<int> pfade;
75         pfade.reserve(graph.size());
76         int kosten = parent[kante].second;
77         int par = kante;
78         pfade.push_back(par);
79         if (par >= n) {
80             bereitsAbgefahren[par]++;
81         }
82
83         do {
84             par = parent[par].first;
85             pfade.push_back(par);
86             if (par >= n) {
87                 bereitsAbgefahren[par]++;
88             } else {
89                 // Nur originale Kanten addieren zur Gesamtlänge des Pfades
90                 kosten += parent[par].second;
91             }
92         } while (par != ZENTRALE);
93
94         // Drehe Pfad um damit er bei der Zentrale beginnt
95         std::reverse(pfad.begin(), pfad.end());
96         pfade.emplace_back(std::make_pair(pfad, kosten));
97
98         // Mache Änderungen an parent vector wieder rückgängig
99         parent[tmp.first] = tmp.second;
100    }
101
102    return pfade;
103}

```

6.2.2 Verbinden der Pfade

```

2 /**
 * Verbindet die gegebenen Pfade solange, bis nur noch 5 übrig sind

```

```

4  * @param pfade Eine Liste an Pfaden
5  * @return Gibt eine Liste mit den verbleibenden 5 Pfaden zurueck
6  */
7  std::vector<std::pair<std::vector<int>, long long>> Algorithmus2::verbindePfade(
8      std::vector<std::pair<std::vector<int>, long long>> &pfade) {
9
10     // Speichert fuer jeden Endknoten, welche Pfade auf diesen enden
11     std::unordered_map<int, std::vector<int>> endKnoten;
12     endKnoten.reserve(pfade.size() * 2);
13
14     // Speichere fuer jeden Pfad den Endknoten
15     for (int i = 0; i < pfade.size(); ++i) {
16         int letzterKnoten = pfade[i].first[pfade[i].first.size() - 1];
17
18         // Speichere Index des Pfades
19         endKnoten[letzterKnoten].emplace_back(i);
20     }
21
22     // Halte Pfade sortiert nach ihrer Laenge
23     auto compare = [] (auto &a, auto &b) {
24         return a.second > b.second;
25     };
26     std::priority_queue<std::pair<std::vector<int>, long long>,
27                         std::vector<std::pair<std::vector<int>, long long>,
28                         decltype(compare)> pq;
29
30     // Verbinde jeweils zwei Pfade mit gleichem Endknoten.
31     for (auto &pair: endKnoten) {
32         if (pair.second.size() < 2) {
33             // Weniger als 2 Pfade mit diesem Endknoten
34             // wird uebersprungen
35             if (pair.second.empty()) continue;
36             pq.push(pfade[pair.second[0]]);
37             continue;
38         }
39
40         if (pair.second.size() % 2 != 0) {
41             // Anzahl ist nicht durch 2 teilbar. Ein Pfad kann nicht verbunden
42             // werden und wird uebersprungen.
43             pq.push(pfade[pair.second[pair.second.size() - 1]]);
44             pair.second.pop_back();
45         }
46
47         for (int i = 0; i < pair.second.size() - 1; i += 2) {
48             // Verbinde jeweils zwei Pfade
49             verbinde(pfade[pair.second[i]], pfade[pair.second[i + 1]]);
50             pq.push(pfade[pair.second[i]]);
51         }
52     }
53
54     while (pq.size() > 5) {
55
56         // Nehme die zwei Pfade mit der kuerzesten Gesamtstrecke
57         auto pfade1 = pq.top();
58         pq.pop();
59         auto pfade2 = pq.top();
60         pq.pop();
61
62         // Verbinde beide Pfade
63         verbinde(pfade1, pfade2);
64
65         // Ergebnis wieder in Priority Queue speichern
66         pq.push(pfade1);
67     }
68
69     pfade.clear();
70     pfade.resize(5);
71
72     unsigned int size = pq.size();
73     for (int i = 0; i < size; ++i) {
74         pfade[i] = pq.top();
75         pq.pop();
76     }

```

```
78     return pfade;
```