G02: Minimaler Steiner Spannbaum

Dokumentation

Lukas Mittermeier, Azim Hasanoglu, Ba Luong Cao

Studiengruppe IF, Bachelor Informatik

Fakultät Informatik und Mathematik

Hochschule
München
University of
Applied Sciences

29.6.2024, Sommersemester 2024

Inhaltsverzeichnis

1 Einführung

Das Ziel dieses Projektes es ist minimale Spannbäume und einen minimalen Steinerbaum zu verstehen und zu finden. Dies wird an einem praxisnahem Beispiel gezeigt.

1.1 Generelle Aufgabe

In Ostbayern soll ein neues Glasfaser-Netz verlegt werden. Dabei sollen 9 Städte einen Anschluss bekommen, wobei München als Anbindung dient also insgesamt 10. 15 weitere Städte stehen optional zur Verfügung als Verzweigungsstellen. Das Kabel verläuft entlang von an Straßen entlang. Dabei entsteht ein Graph mit den Städten als dessen Knoten und die Distanzen zwischen ihnen dienen als Gewichte der Kanten. Unsere Aufgabe deshalb ist es nun einen minimalen Steinerbaum zu finden, welcher mit dem kleinst möglichen Kabelverbrauch alle notwendigen Städte verbinden kann.
Die neuen Städte die angebunden werden sollen sind (Menge R):

1. Augsburg
2. Ingolstadt
3. Regensburg
4. Landshut
5. Straubing
6. Passau
7. Deggendorf
8. Burghausen
9. Weißenburg

Die optionalen Städte für zusätzliche Verzweigungsstellen sind (Menge S):

1. Freising
2. Moosburg a.d. Isar
3. Vilsbiburg
4. Landau a.d. Isar
5. Mainburg
6. Abensberg
7. Schierling
8. Schrobenhausen
9. Erding
10. Pfarrkirchen
11. Osterhofen
12. Eichstätt
13. Neuburg a.d. Donau
14. Kösching
15. Pfaffenhofen a.d. Ilm

2 Algorithmus und Datenvorbereitung

2.1 Algorithmus von Prim

Der Algorithmus von Prim ist ein Algorithmus zur Konstruktion eines minimalen Spannbaums in einem gewichteten Graphen[1]. Der Algorithmus von Prim funktioniert folgendermaßen:

1. Beginne mit einem beliebigen Startknoten und füge ihn zum minimalen Spannbaum hinzu.
2. Markiere diesen Startknoten als besucht und initialisiere eine Prioritätswarteschlange mit allen Kanten, die vom Startknoten ausgehen.
3. Solange es noch unbesuchte Knoten gibt:
   - Entnimm die Kante mit dem kleinsten Gewicht aus der Prioritätswarteschlange.
   - Falls der Endknoten dieser Kante noch nicht besucht wurde:
     - Füge den Endknoten zum minimalen Spannbaum hinzu.
     - Markiere den Endknoten als besucht.
     - Füge alle Kanten, die vom neu hinzugefügten Knoten ausgehen und zu unbesuchten Knoten führen, in die Prioritätswarteschlange ein.
4. Wenn alle Knoten besucht wurden, stoppe den Algorithmus.

Der Algorithmus wählt also schrittweise die Kante mit dem kleinsten Gewicht aus, die einen Knoten aus dem minimalen Spannbaum mit einem unbesuchten Knoten verbindet. Dieser unbesuchte Knoten wird dann zum minimalen Spannbaum hinzugefügt, und der Vorgang wird wiederholt, bis alle Knoten besucht wurden und

der minimale Spannbaum vollständig ist.
Der Algorithmus von Prim garantiert, dass der resultierende Spannbaum minimal ist und dass alle Knoten des ursprünglichen Graphen miteinander verbunden sind, ohne dass es Zyklen gibt. Zur besseren Darstellung und damit man es etwas besser verstehen kann und es vor Augen hat ist hier noch eine Visualisierung des Algorithmus angegeben, die zeigt wie man in einem Graph sich den kürzesten Weg sucht.

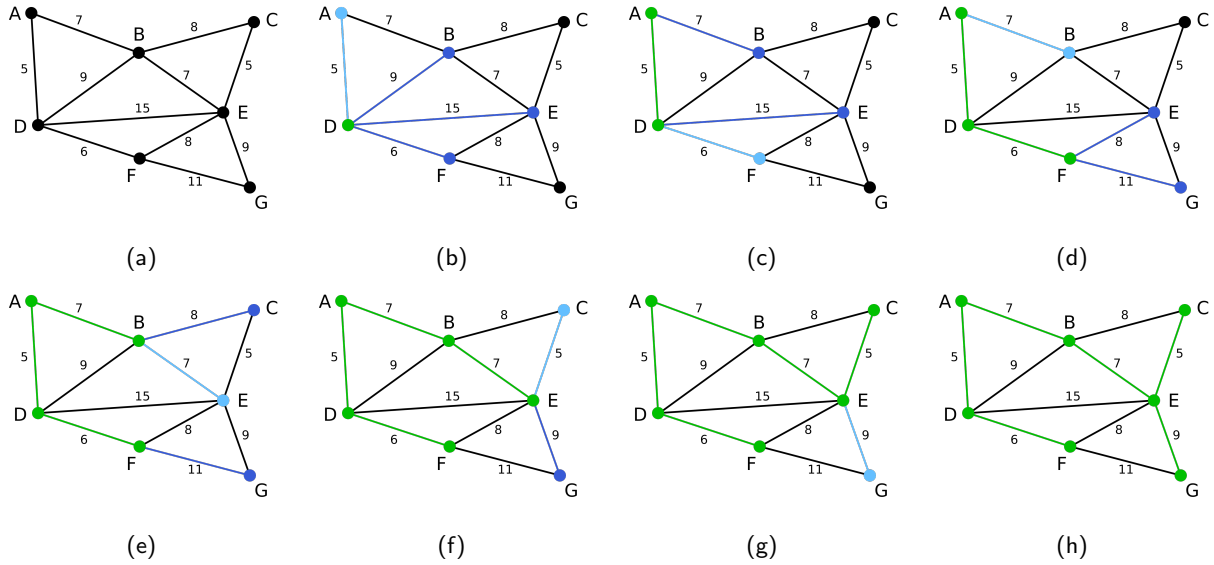

Abbildung 1: Visualisierung des Prim Algorithmus [1]

2.2 Datenvorbereitung

2.2.1 Vorbereitung

Um mit dem Algroithmus von Prim überhaupt arbeiten zu können und Spannbäume aufzustellen brauchen wir Gewichte für die Kanten des Graphen, was unsere Abstände zwischen den verschiedenen Städten sind. Als allererstes muss aber entschieden werden ob man die Luftlinie zwischen zwei Städten nimmt oder den Weg über Straßen. Damit es so realistisch bleibt wie es geht werden die Distanzen über Straßen genommen, da in der Realität solche Kabel auch entlang Straßen verlegt werden. Die beste Möglichkeit um nun an diese Daten zu kommen ist über Online Karten Dienste.

2.2.2 Probleme

Unser ursprünglicher Ansatz war es, zuerst über Code die Webseite luftlinie.org aufzurufen. Bei dieser Website kann man in die Suche seine zwei Städte eingeben und über den HTML-Code werden die bestimmten Distanzen herausgelesen und in einer Karte gespeichert. Das Problem hierbei liegt darin, dass wir auf dieser Website die Luftliniendistanz genommen haben, was in der Theorie zwar richtig, aber nicht sehr praxisnah ist, da die Kabel in der Realität entlang Straßen laufen. Die Website zeigt einem zwar auch die Fahrstrecke an, doch als wir versuchten, auf diesen Wert zuzugreifen, hat es nicht funktioniert. Unsere Vermutung ist, dass unsere Anfrage ankommt, bevor die Strecke berechnet wurde. Auch mit Methoden wie etwas länger zu warten während der Anfrage oder anderes.
Wir haben uns dann weiter erkundigt und sind auf Openrouteservice.org gestoßen. Openrouteservice bietet für die kostenlose Nutzung eine limitierte Möglichkeit an, API-Anfragen zu bearbeiten[2]; so können wir beispielsweise die Koordinate einer Stadt wie München ermitteln.

2.2.3 Openrouteservice API

Es ist sehr wichtig, so wenig API-Anfragen wie möglich zu senden, da die Nutzung für kostenlose Konten begrenzt ist[2]. Für eine spezifische API-Frage kann man sich die API-Dokumentation anschauen[3]. Wir

benutzen für das gesamte Projekt /geocode/search und / directions / driving −car. Die Datenübertragung erfolgt über die request Bibliothek[4].

2.2.4 Aufruf von Koordinaten

Wir definieren zuerst eine Methode namens get_coordinates(staedte). Wenn wir dieser Methode eine Liste von Städtenamen als Strings übergeben, gibt sie uns ein Dictionary zurück. In diesem Dictionary ist der Stadtname der Schlüssel und die Koordinaten (Längengrad und Breitengrad) sind die Werte. Ein Beispiel dafür könnte ' Mnchen ': (11.544467, 48.152126) sein. Um eine Koordinate abzurufen, nutzen wir den API-Befehl https://api.openrouteservice.org/geocode/search. Mit der Bibliothek requests können wir die Antwort im JSON-Format erhalten, dort liegen dann die Informationen. So könnte eine Anfrage für Augsburg lauten: `https://api.openrouteservice.org/geocode/search?api\key=api_key&text=Augsburg`
Zur vereinfachung haben wir einen Flowchart erstellt.

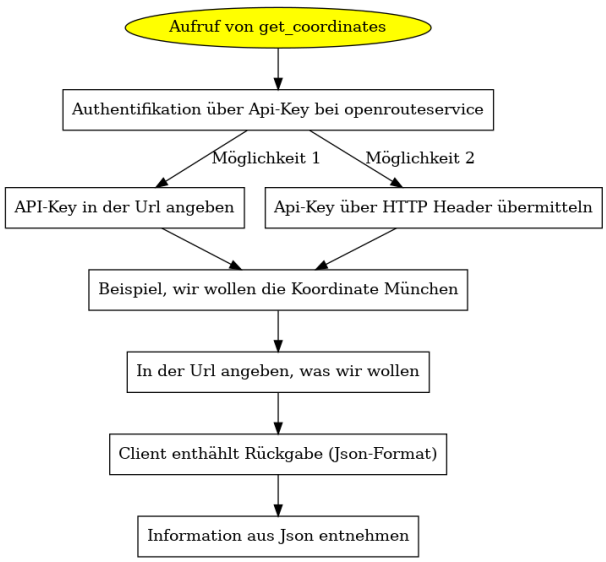

Abbildung 2: Funktionsweise einer API anfrage

2.2.5 Bestimmung der Distanzen

Die bestimmung der Distanzen erfolgt mit der Methode get_distance_between_locations(loc1, loc2). Diese Methode funktioniert so ähnlich wie get_coordinates jedoch, wird eine andere API-Url verwendet. Die Entfernung zwischen Straubing und Augsburg könnte etwa mit folgender Anfrage realisiert werden: `https://api.openrouteservice.org/v2/directions/driving-car?api_key=api_key&start=Straubing&end=Augsburg`

2.2.6 Optimierung der API-Anfragen

Im gesamten Projekt wollen wir die Anzahl der API-Anfragen minimieren, da diese nicht nur limitiert sind, sondern diese auch viel Zeit in Anspruch nehmen. Die Optimierung betrifft folgende Daten.

1. Koordinaten aller Städte, die in R + S sind
2. Alle möglichen Distanzen für die Liste R
3. Alle möglichen Distanzen für die Liste S + R

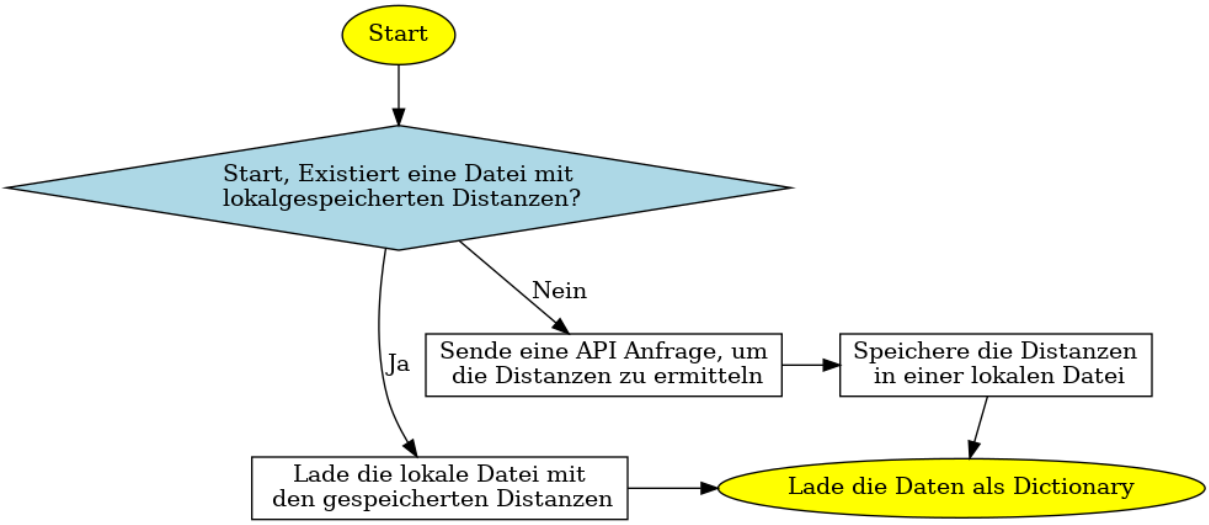

Abbildung 3: Lokale Daten statt API-Anfragen

Wann immer es möglich ist, wollen wir API-Anfragen vermeiden. Wir wollen die Daten, die wir durch die API-Anfrage erhalten, immer in einer Datei abspeichern. Starten wir unsere Rechnungen neu, überprüfen wir, ob eine spezifische Datei mit den Daten existiert. Ist dies der Fall, können wir die HTTP-Anfrage überspringen. D.h., wir wollen auf dem Flowchart immer den "Ja-Fall"durchlaufen.
Für das Speichern und Laden verwenden wir die Pickle-Bibliothek. Diese Bibliothek unterstützt unter anderem das Speichern von zweier Tupel als Schlüssel für ein Dictionary. Der Nachteil von Pickle ist jedoch, dass die Daten nicht in einem lesbaren Format gespeichert werden. Möchte man die Anfragen neu starten, genügt es, die lokalen .pkl-Dateien zu löschen. Mit den .pkl Dateien sparen wir uns etwa 5-10 Minuten pro Compile, die sonst für API-Anfragen nötig wären.

2.3 Verifizierung per Dreiecksungleichung

Damit wir aber auch wissen das wir nun alle kürzesten Strecken für den Graphen benutzen brauchen wir die sogenannte Dreiecksungleichung. Diese sagt aus das eine direkte Strecke zwischen u und v immer kürzer ist als die Strecke über einen dritten Punkt w. Die Dreiecksungleiung lautet:

$$d(u,v) \leq d(u,w) + d(w,v) \text{ mit } u,v,w \in V$$

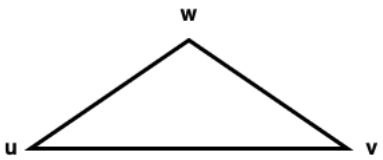

Abbildung 4: Dreiecksungleichung Graph

Im Code haben wir drei Städte aus V in die Formel als u, v, w eingesetzt und überprüfen lassen ob die Relation stimmt. Dies haben wurde dann noch über eine Schleife mit jeder möglichen Kombination aus drei Städten gemacht. Eine Problematik die aufgetreten ist lautet das dadurch das wir die Strecken von der API entlang Straßen nehmen also es als Autofahrt betrachten wird, es bei ein paar Städten passiert das die direkte Strecke zwischen zwei Städten einn extremen Umweg nimmt und so die Dreiecksungleichung nicht mehr stimmt. Um dieses Problem zu lösen haben wir uns entschieden bei Strecken mit extremen Umweg stattdessen die Luftlinie zu benutzen, wodurch zwar ein kleiner Fehlerbereich entsteht jedoch ist dieser eher gering bei Betrachtung des ganzen Graphens. Um die Luftlinie zu berechnen haben wir die haversine Formel benutzen mit der man mit zwei Koordinaten die Luftlinie bekommt.[15] Dies hätte zwar auch eventuell mit der API funktionieren

können, aber wir wollten uns zusätzliche API-Anfragen sparen. Die Logik im Code, um die Fehler zu beheben sieht so aus.

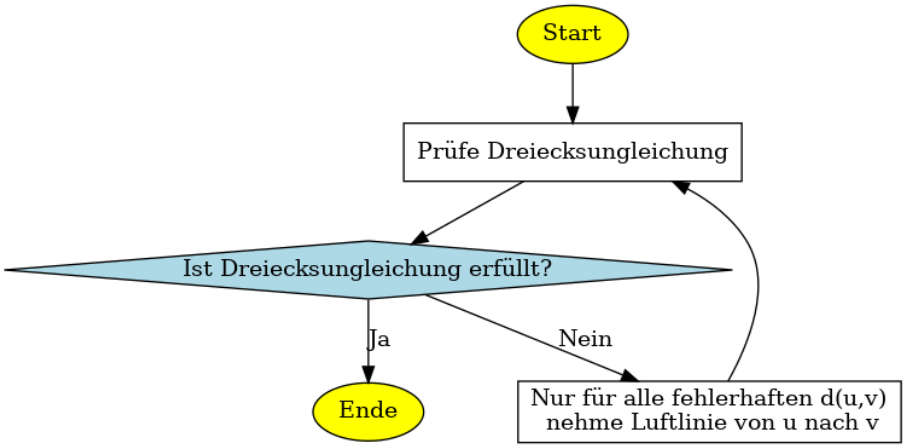

Abbildung 5: Fehlerbehebung in der Dreiecksungleichung

3 Minimaler Spannbaum in R

In diesem Abschnitt implementieren wir den Prim Algorithmus als Code und wenden ihn mit unseren Distanzen als Gewichte an. Dieser wird jedoch zuerst nur mit den notwendigen 10 Städten die mit Glasfaser Kabel angebunden werden sollen, durchgeführt. Um zu überprüfen ob dieser Graph denn auch nun stimmt benutzen wir eine bereits implementierte Version des Prim Algorithmus ,welchen man über den SageMath Befehl *.min_spanning_tree()* aufrufen kann und vergleichen dessen Graph mit unserem. Der Graph aus dem eigen implementierten Algorithmus sieht wie folgt aus:

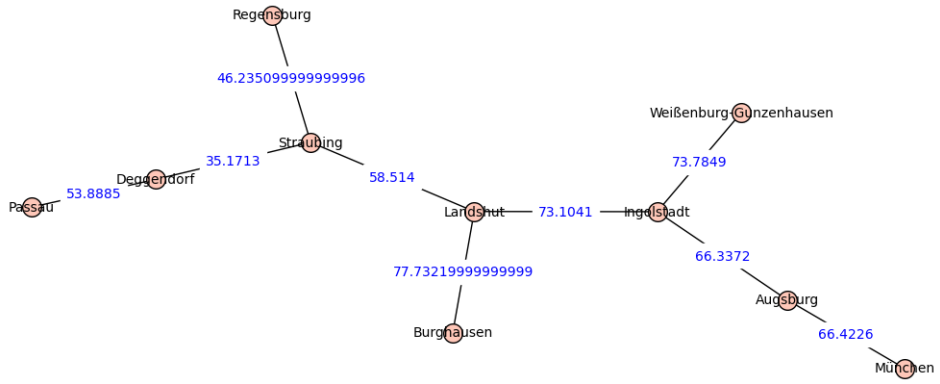

Abbildung 6: Minimaler Spannbaum in R

3.1 Verifizierung zusätzlich über SageMath

Um zu verifizieren ob unser Algorithmus richtig ist, können wir gegen prüfen mit dem Algorithmus der bereits in SageMath implementiert ist[6]. Da die Anzahl der Knoten überschaubar ist, können wir nach dem rendern die Form und die Gewichte grafisch vergleichen.

```
Graph(G.min_spanning_tree(algorithm='Prim_Boost', weight_function=weight))
```

Wir können beobachten, dass der gleiche Graph gerendert wird. Das ist für uns ein Zeichen das unser Algorithmus logisch richtig ist.

4 Minimaler Steinerbaum in $R \cup S$

4.1 Steinerbaumproblem

Das Steinerbaumproblem ist eine ähnliche Fragestellung zu dem minimalen Spannbaum, da die Aufgabe grundsätzlich darin liegt in einem Graphen nach dem kürzesten Weg bei angegeben Knoten zu suchen. Der große Unterschied was es zu seinem eigenen Problem macht ist das weitere optionale Knoten, namens Steinerpunkte, hinzugefügt werden die als zusätzliche Verzweigungen im Graphen dienen um die insgesamte Strecke zwischen den Punkte weiter zur verkürzen. Jedoch müssen diese Steinerpunkte nicht zwingend mit einbezogen werden sondern stehen als Hilfe zur Verfügung. Ein minimaler Spannbaum dieser Art wird als minimaler Steinerbaum bezeichnet und besitzt höchstens $|R| - 2$ Steinerpunkte. Dieses Steinbaumproblem können wir auf unser Glasfaser Netz anwenden um die 15 zusätzlich optionalen Städte als Steinerpunkte mit einzubinden und somit einen noch streckeneffizienteren Graphen zu finden.

4.2 Finden des minimalen Steinerbaums

Es gibt vermutlich viele Wege für den minimalen Steinerbaum. Den Ansatz, welchen wir hier anwenden werden ist eine *Brute-Force-Methode*[7]. Grundlegend besteht dieser Ansatz also daraus das von allen Kombinationen der optionalen Städte ein minimalen Spannbaum mit dem Prim Algorithmus berechnet wird und dies solange bis man den richtigen findet. Daraufhin werden folgende Schritte ausgeführt:

1. Eine Anzahl von höchstens 8 Städte aus der Menge S wird mit der Menge R zusammengefügt zu V
2. Ein minimaler Spannbaum mit dem Prim Algorithmus wird von der Menge V erstellt und dessen Graphen und insgesamter Streckenwert wird zwischengespeichert
3. Ein neuer Spannbaum wird erneut mit Schritt 1 und 2 erstellt
4. Diese beiden Spannbäume werden miteinander verglichen, dies bedeutet das geschaut wird wer den kleineren Streckenwert besitzt
5. Der kleinerer Wert und Graph wird wieder zwischengespeichert
6. Schritte 1-5 wird über eine Schleife wiederholt bis alle Kombinationen von 1-8 optionalen Städten einen Graphen bekommen haben und dieser verglichen wurde

Zur vereinfachten Darstellung der Schritte ein Flowchart, welcher es etwas übersichtlicher machen sollte:

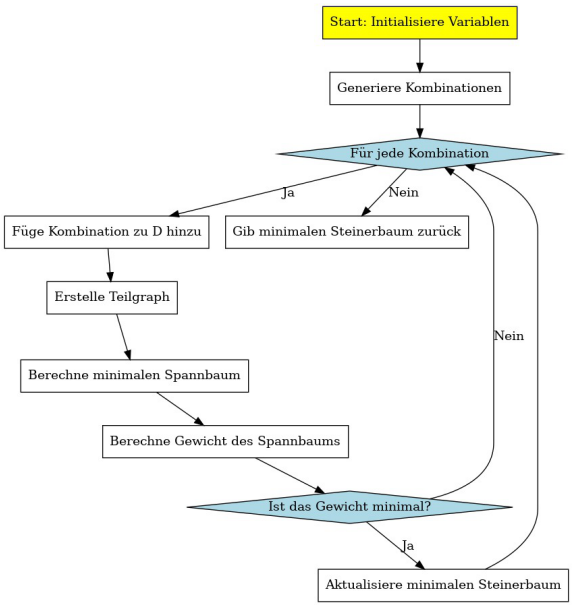

Abbildung 7: Schritte zum Finden eines minimalen Steinerbaums

Am Ende nachdem all diese Schritte ausgeführt wurden ist der übrig gebliebene Graph der gesuchte minimale Steinerbaum mit zusätzlichen Städten und dem kleinsten Kabelverbrauch den es geben kann bei den angegeben Städten. Der entstandene minimaler Steinerbaum sieht wie folgt aus:

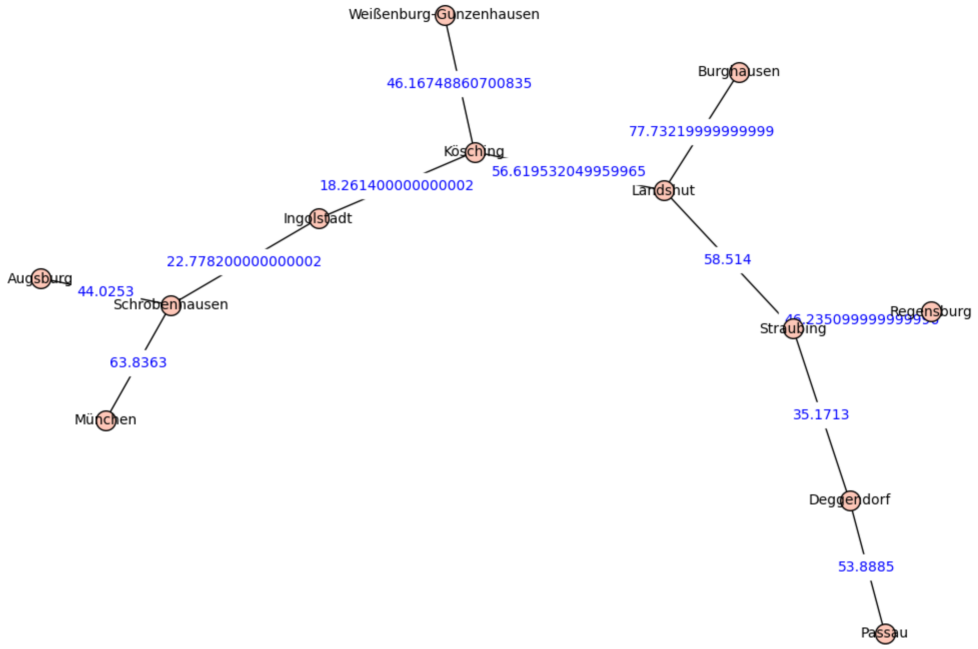

Abbildung 8: Minimaler Steinerbaum in V

5 Diskussion

5.1 Messungen im Steiner Algorithmus

Wir haben beim Algorithmus für den minimalen Steinerbaum Bereiche, in denen wir Messungen vornehmen. Unter anderem zeichnen wir auf, wie lange es dauert, n-Knoten zu berechnen. Hierbei erhalten wir folgenden Graphen.

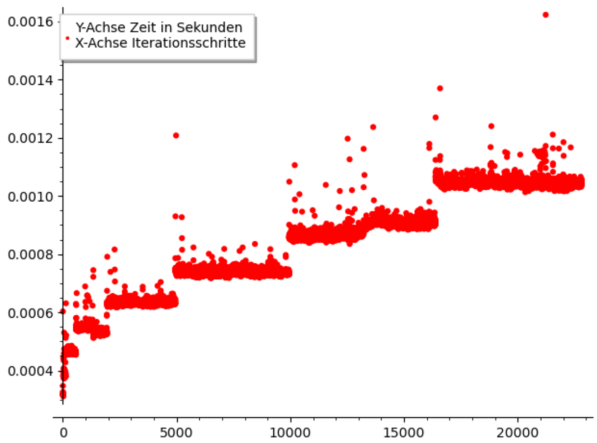

Abbildung 9: Dauer pro Iterationsschritt

Nun betrachten wir unsere Messungen. Wir beobachten, dass sich die Zeit für die Berechnung erhöht, sobald sich die Anzahl der von D im Algorithmus für den minimalen Steinerbaum erhöht. D sind die Großstädte +

der(die) Steinerpunkt(e). Die Messungen ergeben einen treppenartigen Graphen. Wir interpretieren das wie folgt:

1. Jede Ansammlung von Punkten (eine Stufe) repräsentiert eine Gruppe von Kombinationen mit der gleichen Anzahl von Knoten.
2. Der Übergang von einer Stufe zur nächsten zeigt den zusätzlichen Rechenaufwand, der durch das Hinzufügen weiterer Knoten entsteht.
3. Die Höhe jeder Stufe gibt die durchschnittliche Zeit für die Berechnung aller Kombinationen dieser Knotenmenge an.

Die Ausreißer könnten auf spezifische Kombinationen von Knoten zurückzuführen sein, so dass der Aufruf von prim_algorithm(G_Teil) im Algorithmus für den minimalen Steinerbaum etwas mehr Zeit benötigt.

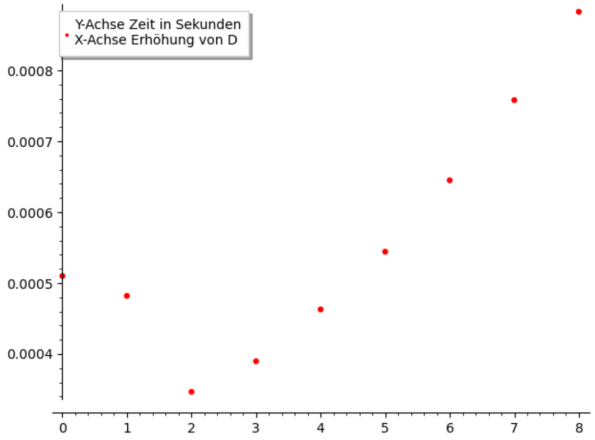

Abbildung 10: Ein Versuch die Mittelwerte einer Stufe zu ermitteln

Wir haben außerdem versucht, die Mittelwerte jeder Stufe zu bilden. Wir beobachten einen fast linearen Anstieg, bis auf die Werte an den Stellen $x = 0$ und $x = 1$, diese könnte auf das Initialisieren der Methode und Dictionaries zurückführen. Der Anstieg insgesamt erscheint uns logisch, da der Prim-Algorithmus, der innerhalb des Algorithmus für den minimalen Steinerbaum verwendet wird, für jedes Element, das zu D hinzukommt, mehr Rechenaufwand benötigt.

5.1.1 Idee zur Bestimmung des Aufwands für höhere Werte

Da sich die Kombinationen sehr schnell erhöhen, ist das Testen der Messung der Zeit extrem aufwendig. Erhöht man die Anzahl der Städte in R bereits um 6, hätten wir $\binom{25+6}{2}$ Kombinationen allein für die API-Anfragen. Deshalb versuchen wir es mit einem anderen Ansatz. Setzen wir für die obere Grenze 100 ($|R| = 100$) und für das n im Binomialkoeffizienten die Zahl 200 ($|S| = 200$), erhalten wir für $\sum_{i=0}^{100} \binom{200}{i}$ sehr viele Kombinationen. Wir können zeigen, dass die Berechnung für diese Anzahl an Kombinationen extrem lange dauert.

1. Dazu können wir eine einfache For-Schleife ohne zusätzliche Logik verwenden, d.h., wir messen die reine Zeit, die eine For-Schleife benötigt, und multiplizieren sie mit der Anzahl der Kombinationen. Für $\sum_{i=0}^{100} \binom{200}{i}$ erhalten wir $8{,}487432795 \cdot 10^{59}$ Kombinationen.
2. Nun messen wir die Zeit für eine For-Iteration. Vermutlich können hier Abweichungen je nach Programmiersprache entstehen. In SageMath beträgt die Dauer etwa $4{,}4 \cdot 10^{-14}$ Sekunden.
3. Wir können diese Zahl nun mit der Anzahl der Kombinationen multiplizieren und erhalten $3{,}73447042961321 \cdot 10^{55}$ Sekunden. Das sind umgerechnet $1{,}18419280492555 \cdot 10^{48}$ Jahre.

Hier erkennt man, wie aufwendig das Finden eines minimalen Steinerbaums ist. Selbst wenn man hier optimistisch ist und mit 1 Nanosekunde rechnet, wäre die benötigte Zeit für das Finden des minimalen Steinerbaums immer noch viel zu hoch.

Nun können wir eine Vermutung anstellen, warum das so ist. Wenn man n schrittweise in $\sum_{i=0}^{m} \binom{n}{i}$ erhöht, wobei $n, m \in N$, $n \geq m$, nimmt die Anzahl der Kombinationen exponentiell zu (d.h. n könnte z.B. $|S|$ sein und m unsere Großstädte $|R|$ wie mit Beispielwerten aus Teilaufgabe 3). Der exponentielle Zuwachs liegt vermutlich an der Fakultät im Binomialkoeffizienten $\binom{n}{i} = \frac{n!}{i!(n-i)!}$. Durch die Fakultät wird n im Zähler sehr groß. Lässt man jetzt beispielsweise i gegen $\lfloor \frac{n}{2} \rfloor$ laufen, erhält man eine sehr große Zahl für den Binomialkoeffizienten.

5.2 Vergleich der beiden Spannbäume

Ein weiterer wichtiger Punkt ist der Vergleich der beiden Spannbäume. Denn nach der ganzen Arbeit, die Graphen zu berechnen und optionale Städte einzubeziehen, wäre es natürlich schade, wenn am Ende der minimale Steinerbaum kaum Effizienzvorteile gegenüber dem „einfacheren“ minimalen Spannbaum hätte. Um dies zu vergleichen, addiert man ohne Weiteres alle Kantengewichte zusammen und erhält so die gesamte Kabellänge des Graphen.
Bei dem **minimalen Spannbaum** bekommt man ein Ergebnis von: $657,63\,\mathrm{km}$ an Glasfaserkabel
Bei dem **minimalen Steinerbaum** bekommt man ein Ergebnis von: $523,23\,\mathrm{km}$ an Glasfaserkabel
Das bedeutet, dass wir mit unserem neuen Steinerbaum ganze 134,40 km Glasfaserkabel uns sparen, was 20% entspricht. 20% sind schon eine erhebliche Verbesserung, vor allem bei einem Projekt dieser Größe kann dies wahrscheinlich die Kosten stark verringern. was schon auch erheblich die Kosten beeinflussen kann die das ganze Projekt kostet.

5.3 Erweiterung

5.3.1 Koordinaten anwenden

Eine mögliche Erweiterung wäre beispielsweise die Anwendung der Koordinaten auf dem SageMath-Graphen. Aktuell ist es so, dass die Knoten automatisch positioniert werden. Wir wollen jedoch für die bessere Darstellung die Knoten so positionieren, als würde man die Städte (d.h. die Knoten) aus einer Landkarte ablesen. Die Gewichte und unsere Berechnungen bleiben hierdurch unberührt, wir verändern den Graphen nur visuell.

5.3.2 Auswahl der Routen

Wir können bei den API-Beispielen von openrouteservice eine URL so modifizieren, dass nicht mehr die Fahrstrecken, sondern beispielsweise die Fahrradwege oder die Fußwege berücksichtigt werden [5]. Dies könnte ein weiterer Optimierungsgrund sein, da Fußgängerwege möglicherweise kürzer sein könnten.

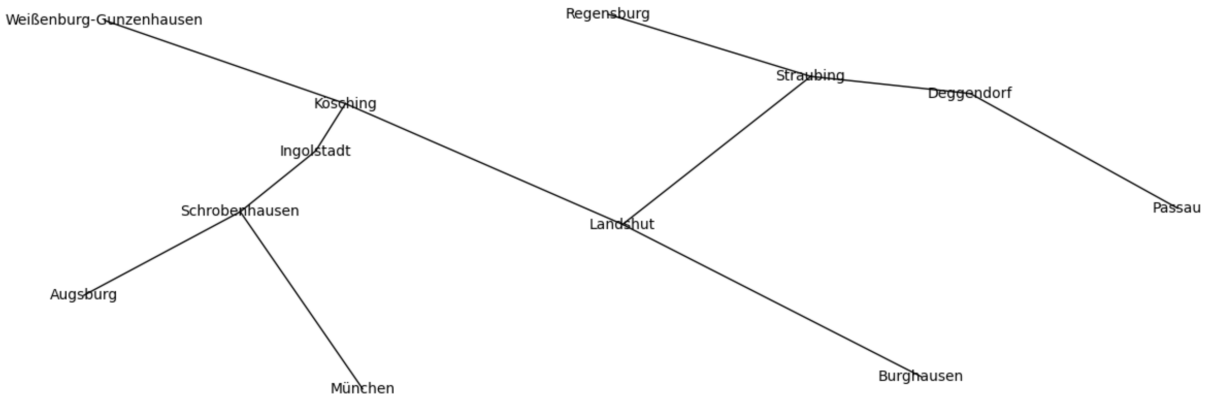

Abbildung 11: Mögliche Form eines optimalen Glasfasernetzes

6 Bibliotheken

6.1 datetime

1. Arbeiten mit Zeitangaben[8].
2. In unserem Projekt nötig für Dauer von Berechnungen

6.2 os

1. Interaktion mit dem Betriebssystem[9].
2. Im Projekt verwendet, um beispielsweise zu Überprüfen, ob eine Datei existiert.

6.3 pickle

1. Serialisieren und Deserialisieren von Python-Objekten[10].
2. Im Projekt verwendet, um API-Anfragen zu minimieren

6.4 itertools

1. Das Modul standardisiert eine Kernsammlung schneller, speichereffizienter Werkzeuge, die nützlich sind, entweder einzeln oder in Kombination[14].
2. Im Projekt verwendet für die Erstellung von Kombinationen, die in einer Liste gespeichert werden Können

6.5 requests

1. Requests erlaubt es, HTTP/1.1 Anfragen äußerst einfach zu senden[11].
2. Im Projekt verwendet für das Arbeiten mit der API (z.B. GET)

6.6 time

1. Diese Bibliothek bietet verschiedene zeitbezogene Funktionen[12].
2. Einsatz für Cooldowns in userem Projekt, beispielsweise für zeitlich limitierte API-Anfragen

6.7 math

1. Dieses Modul bietet Zugriff auf die mathematischen Funktionen[13]
2. Im Projekt verwendet für fehlerhafte Dreiecksungleichungen
3. radians implementiert Bogenmaß
4. cos implementiert Kosinus
5. sin implementiert Sinus
6. asin implementiert Arkussinus
7. sqrt implementiert Wurzel

7 Literatur

[1] Algorithmus von Prim. Wikipedia. https://de.wikipedia.org/wiki/Algorithmus_von_Prim. Abrufdatum: 28. Juni 2024.

[2] OpenRouteService Tarife. https://openrouteservice.org/plans/. Abrufdatum: 28. Juni 2024.

[3] OpenRouteService API Dokumentation. https://openrouteservice.org/dev/#/api-docs/. Abrufdatum: 28. Juni 2024.

[4] requests.get() Funktion in Python. https://www.w3schools.com/PYTHON/ref_requests_get.asp. Abrufdatum: 28. Juni 2024.

[5] OpenRouteService API Anfragen. https://openrouteservice.org/dev/#/api-docs/v2/directions/{profile}/get. Abrufdatum: 28. Juni 2024.

[6] Prim-Boost Algorithmus in SageMath. https://doc.sagemath.org/html/en/reference/graphs/sage/graphs/generic_graph.html. Abrufdatum: 28. Juni 2024.

[7] Brutforceproblem bei YouTube. https://www.youtube.com/watch?v=BG4vAoV5kWw, Minute 1:06:00. Abrufdatum: 28. Juni 2024.

[8] Python datetime Bibliothek. https://docs.python.org/3/library/datetime.html. Abrufdatum: 28. Juni 2024.

[9] Python os Bibliothek. https://docs.python.org/3/library/os.html. Abrufdatum: 28. Juni 2024.

[10] Python pickle Bibliothek. https://docs.python.org/3/library/pickle.html. Abrufdatum: 28. Juni 2024.

[11] Python requests Bibliothek. https://requests.readthedocs.io/en/latest/. Abrufdatum: 28. Juni 2024.

[12] Python time Bibliothek. https://docs.python.org/3/library/time.html. Abrufdatum: 28. Juni 2024.

[13] Python math Bibliothek. https://docs.python.org/3/library/math.html. Abrufdatum: 28. Juni 2024.

[14] Python itertools Bibliothek. https://docs.python.org/3/library/itertools.html. Abrufdatum: 28. Juni 2024.

[15] Python itertools Bibliothek. https://stackoverflow.com/questions/4913349/haversine-formula-in-python-bearing-and-distance-between-two-gps-points. Abrufdatum: 28. Juni 2024.