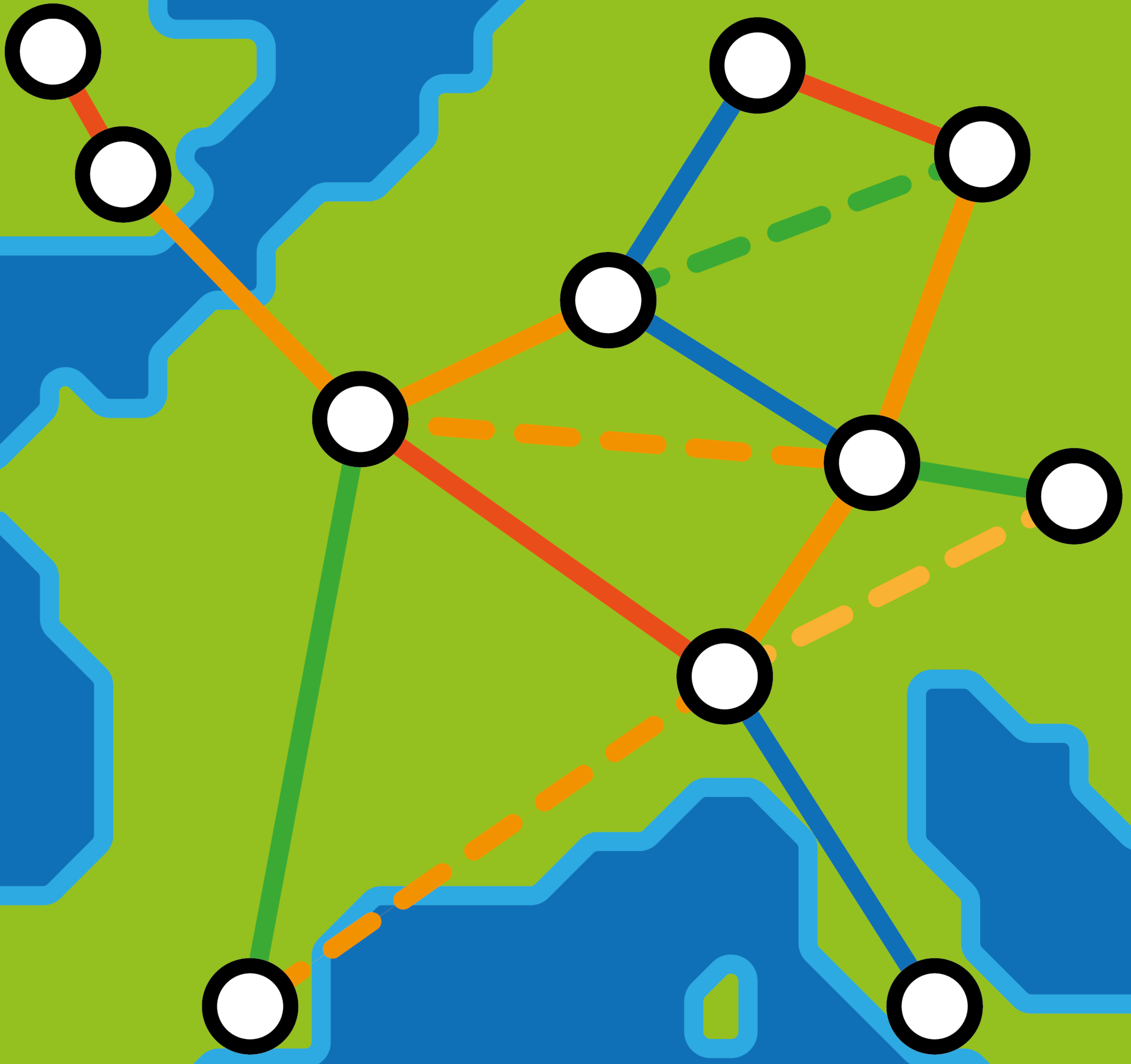
Voll vernetzt!

Transportnetzwerke mit minimalen Kosten und maximaler Effizienz



Von Donald Kobbelt

Voll vernetzt!

von Donald Kobbelt (8c)

1. Motivation und Fragestellung 1
2. Hintergrund und theoretische Grundlagen 2
3. Mathematische Modellierung des Problems 5
4. Optimierungsmethode 8
5. Ergebnisse 10
6. Ergebnisdiskussion, Fazit und Ausblick 14
7. Quellen- und Literaturverzeichnis 16

Zusammenfassung

Weil viele Leute zu ihrer Arbeit pendeln und wir immer mehr Sachen im Internet bestellen, nimmt der Verkehr von Personen und Waren immer mehr zu. Daran wird sich in Zukunft wohl auch nichts ändern. Es ist also sehr wichtig, Transport-Netzwerke genau zu planen, damit man möglichst schnell und günstig von einer Stadt in eine andere reisen kann.

In meinem Projekt habe ich daher versucht Netzwerke zu optimieren. Dazu habe ich mir überlegt, wie man in einem Python-Programm Netzwerke erzeugen kann, um darin dann Verbindungen zu berechnen und zu optimieren.

Wenn man ein Netzwerk plant, dann möchte man sowohl die Baukosten als auch die Reisezeit minimieren. Diese beiden Ziele widersprechen sich aber eigentlich, denn für eine kürzere Reisezeit benötige ich zusätzliche Verbindungen (Schienen oder Straßen), was zu erhöhten Baukosten führt. D. h. ich muss einen guten Kompromiss zwischen Kosten und Reisezeit finden. Das mache ich mit der sogenannten Pareto-Optimierung, bei der ich neue Verbindungen in ein Netzwerk einfüge oder alte Verbindungen lösche, um ein möglichst gutes Verhältnis von Nutzen (kurze Reisezeit) und Kosten zu erhalten.

1. Motivation und Fragestellung

Um den Klimawandel aufzuhalten, sollten wir alle weniger mit dem eigenen Auto fahren, sondern lieber mit der Bahn oder in Zukunft vielleicht sogar mit der Hyperloop. Aber um immer mehr Personen schnell und zuverlässig befördern zu können, benötigt die Bahn ein effizientes Schienennetz.

Zurzeit befindet sich die Deutsche Bahn am Anfang einer grundlegenden Sanierung und Erweiterung. Damit sind extrem hohe Kosten verbunden. Beispielsweise soll die *Riedbahn* zwischen Frankfurt und Mannheim für 1,3 Milliarden € saniert werden [1]. Das sind Kosten von 18,6 Millionen € pro Kilometer! Der Neubau von Schienenverbindungen ist wahrscheinlich noch teurer. Da lohnt es sich, die Verbindungen zu optimieren. Doch wie könnte und sollte ein neues Schienennetz aussehen und wie müsste man so ein Netzwerk konzipieren, so dass es zu schnellen Verbindungen und gleichzeitig zu günstigen Baukosten führt?

Genau das möchte ich in diesem Projekt herausfinden: Wie kann man überhaupt Netzwerke mit einem Programm berechnen und optimieren? Nach welchen Kriterien kann ich die Qualität eines Netzwerks bestimmen und wie finde ich dann Netzwerke, die diese Kriterien erfüllen? Bei meiner Suche im Internet habe ich gelesen, dass die Optimierung von Netzwerken ein kompliziertes, aber sehr aktuelles Forschungsgebiet ist, wo Mathematiker, Informatiker und auch Wirtschaftswissenschaftler zusammenarbeiten.

Mein Ziel ist es, einen Algorithmus zu entwickeln, zu implementieren und zu testen, mit dem man Netzwerke nach bestimmten Kriterien optimieren kann. Ich möchte dabei sowohl die Baukosten und Reisezeiten als auch die Passagierzahlen berücksichtigen. Mein Algorithmus soll allgemein für beliebige Transport-Netzwerke verwendbar sein und ich möchte ihn in Python programmieren. Um auch konkrete und praktische Ergebnisse zu produzieren, werde ich den Algorithmus auf die Berechnung eines Schienennetzwerkes zwischen den **38** größten deutschen Städten, zwischen den **20** größten französischen Städten und sogar zwischen den **234** wichtigsten Städten in ganz West-Europa anwenden.

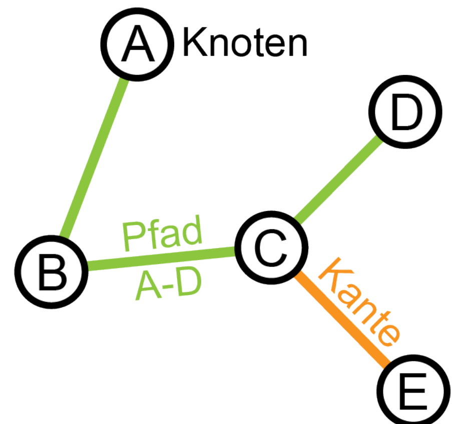
In meinem Projekt soll es aber nur um die *Planung* eines Netzwerkes gehen, was sich schon als ziemlich kompliziert herausstellt. Ich untersuche *nicht*, wie man einen Fahrplan berechnet, also wo und wann wie viele Züge fahren sollen. Dafür müsste ich zusätzlich Informationen berücksichtigen, zu welcher Tageszeit die Leute reisen und wo sie umsteigen. Diese Daten standen mir aber für das Projekt leider nicht zur Verfügung.

2. Hintergrund und theoretische Grundlagen

In diesem Kapitel fasse ich die mathematischen Grundlagen zusammen, wie man Netzwerke speichern und verarbeiten kann. Außerdem erkläre ich wie man kürzeste Verbindungen in Netzwerken suchen kann, wie man Netzwerke nach mehreren Kriterien optimiert und wie man Netzwerke geometrisch konstruieren kann.

Graphen [2] [3]

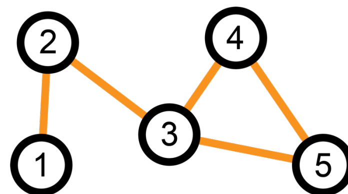
In der Datenstruktur, mit der ich arbeite, werden Städte als Knoten bezeichnet (A bis E in der Zeichnung), da diese meist mehrere Verbindungen zu anderen Knoten besitzen. Zwischen den Knoten befinden sich die Kanten, welche jeweils zwei Knoten direkt miteinander verbinden (grün und orange in der Zeichnung). Jede Kante hat eine Länge, die genau der Distanz zwischen den beiden End-Knoten entspricht.



Ein Pfad ist ein Weg, der zwei nicht direkt benachbarte Knoten über eine oder mehrere Kanten miteinander verbindet (grün in der Zeichnung). Die Summe der Längen der Kanten entlang des Pfades ist die Länge des Pfades. Zwischen zwei Knoten kann es mehrere verschiedene Pfade geben, aber mich interessiert immer nur der kürzeste. Kürzeste Pfade können keine Schleifen enthalten. Wenn ein kürzester Pfad von Knoten A nach Knoten C über einen Knoten B verläuft, dann ist das Teilstück dieses Pfades, welches von Knoten A bis Knoten B verläuft auch der kürzeste Pfad von Knoten A nach Knoten B. Das gleiche gilt für das andere Teilstück des Pfades von Knoten B nach Knoten C.

In Python speichert man die Kanten eines Graphen in einer Verbindungsmatrix A. In dieser Matrix steht in der i-ten Zeile und der j-ten Spalte eine 1, wenn die beiden Knoten i und j mit einer Kante direkt verbunden sind. Da meine Verbindungen (Gleise) in beiden Richtungen befahren werden können, steht dann immer auch in der j-ten Zeile und der i-ten Spalte eine 1. Wenn zwei Knoten *nicht* direkt verbunden sind, steht eine 0 in der Matrix. Zum Beispiel:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$



In dem zugehörigen Python Programm wird eine leere Verbindungsmatrix für einen Graphen mit `n` Knoten durch `A = np.zeros((n,n))` definiert. Danach werden alle Verbindungen mit `A[i][j] = 1` und `A[j][i] = 1` eingetragen.

Andere Informationen über Kanten oder Knotenpaare kann man auch in einer Matrix speichern, wie zum Beispiel die Passagierzahlen. In der Matrix P bedeutet dann der Eintrag `P[i][j] = 100`, dass 100 Passagiere von Knoten `i` nach Knoten `j` reisen.

Bäume und der Kruskal-Algorithmus

Ein Baum ist eine Verbindungsmöglichkeit von mehreren Knoten mit Verzweigungen aber ohne einen Kreislauf zu erzeugen, sprich es gibt zwischen zwei Knoten immer nur einen eindeutigen Pfad. Man nennt einen Baum einen Spannbaum, wenn er *alle* Knoten eines Graphen miteinander verbindet. Ein minimaler Spannbaum ist ein Spannbaum, bei dem die Summe der Längen aller vorhandenen Kanten möglichst gering ist. So einen minimalen Spannbaum kann man mit dem Kruskal-Algorithmus [4] berechnen.

Dieser funktioniert wie folgt: Zuerst werden alle Kanten (von jedem Knoten zu jedem Knoten) der Länge nach sortiert. Danach werden die Kanten von klein bis groß nacheinander aktiviert. Dies geschieht, bis alle Knoten miteinander verbunden sind. Allerdings werden Kanten, die einen Kreislauf verursachen würden, übersprungen, da diese das Kriterium für einen minimalen Spannbaum verletzen.

Kürzeste Pfade und der Floyd Warshall-Algorithmus

Für die Bewertung des Schienennetzes muss ich ausrechnen, wie lange man von einem Knoten `i` zu einem Knoten `j` fahren muss, das heißt ich benötige für jedes Knotenpaar `[i][j]` die Länge des kürzesten Pfades. Diese kürzesten Längen speichere ich wieder in einer Matrix S, d.h. `S[i][j] = S[j][i]` ist die Länge des kürzesten Pfades zwischen i und j.

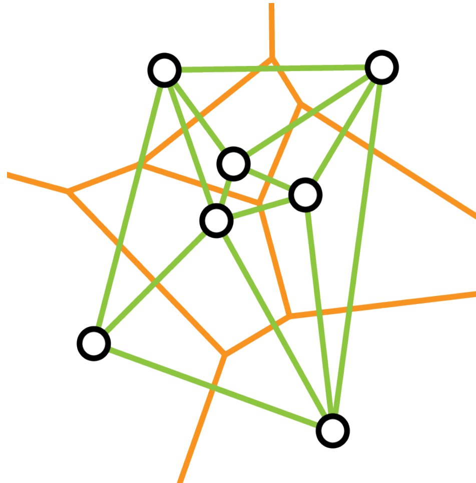
Mit dem Floyd Warshall-Algorithmus [5] kann man diese Matrix S berechnen. Der Algorithmus nutzt dabei aus, dass jedes Teilstück von einem kürzesten Pfad auch selbst ein kürzester Pfad ist. Das heißt, wenn der kürzeste Pfad von `i` nach `j` über den Knoten `k` führt, dann ist die Länge dieses Pfades gleich der Summe der kürzesten Pfade von `i` nach `k` plus von `k` nach `j`. Das Vorgehen besteht nun darin, für jeden möglichen Zwischenknoten `k` zu testen, ob der Weg von `i` über `k` nach `j` der kürzeste ist.

Der Floyd Warshall-Algorithmus beginnt damit in der Matrix S nur die direkten Verbindungen (ohne Zwischenknoten) zu speichern. Wenn also in der Verbindungsmatrix an der Stelle `A[i][j]` eine `1` steht, dann wird in `S[i][j]` die Länge der entsprechenden Kante gespeichert. Wenn `A[i][j]` gleich `0` ist, dann speichere ich in `S[i][j]` eine sehr große Zahl, die viel größer ist als alle möglichen Distanzen, z. B. 1.000.000.

In den nächsten Schritten versucht der Algorithmus kürzere Verbindungen zwischen Knoten zu finden, indem er zuerst vom Knoten `i` zu einem Zwischenknoten `k` und dann weiter zum Knoten `j` läuft. Die *bisher* kürzesten Verbindungen von i nach j, von i nach k und von k nach j

sind in der Matrix S gespeichert. Wenn also `S[i][k] + S[k][j]` kleiner ist als `S[i][j]`, dann ist der Pfad über `k` kürzer und man speichert den neuen Wert in `S[i][j]`. Dies macht man in einer *dreifachen* Schleife über alle `k` und alle Paare `[i][j]`.

Delaunay Triangulierung

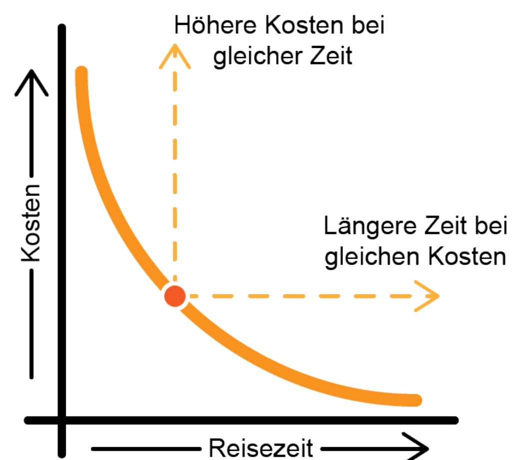


Die Delaunay Triangulierung [6] basiert auf dem Voronoi Diagramm [7]. In einem Voronoi Diagramm gibt man eine Menge von Zentren vor und berechnet dann für jedes Zentrum ein Voronoi-Gebiet. Dieses Voronoi Gebiet besteht aus allen Punkten, die näher an diesem Zentrum liegen als an irgendeinem anderen Zentrum. Das Bild links zeigt ein Voronoi Diagramm (in orange). In meinem Projekt sind die Städte die Zentren, für die ein Voronoi Diagramm als „Nachbarschaft" berechnet wird.

Für die Delaunay Triangulierung werden nun die Zentren, die eine gemeinsame Grenze haben, miteinander durch eine Kante verbunden. Eine Delaunay Triangulierung kann man daher als die natürliche geometrische Art verstehen, wie man Zentren mit seinen Nachbarn verbindet. Der Algorithmus zur Berechnung einer Delaunay Triangulierung ist relativ kompliziert, daher habe ich die `MathPlotLib`-Erweiterung für Python verwendet mit der Delaunay Triangulierungen automatisch berechnet werden können.

Pareto-Optimierung [8]

Für die Optimierung der Netzwerke sind zwei Kriterien wichtig, nämlich die Baukosten und die Reisezeiten. Je mehr direkte Verbindungen zwischen den Städten gebaut werden, desto schneller kann man reisen (weniger Umwege über andere Städte), aber gleichzeitig steigen auch die Baukosten. Daran kann man erkennen, dass sich die beiden Ziele, kurze Reisezeiten und geringe Baukosten, gegenseitig widersprechen.



Bei so einem Problem, mit widersprechenden Zielkriterien, gibt es nicht nur eine optimale Lösung, sondern es gibt mehrere Lösungen, für die alle gelten, dass sich die Reisezeit bei gleichbleibenden Kosten nicht weiter reduzieren lässt und gleichzeitig die Kosten bei gleichbleibender Reisezeit nicht gesenkt werden können. Das Schaubild rechts veranschaulicht diese Situation. Man spricht dann von einer Pareto-Optimierung, bei der versucht wird,

die Pareto-Kurve („Pareto-Front“) so weit wie möglich nach unten links im Schaubild zu schieben. An der Pareto-Kurve kann man ablesen, welche Reisezeit sich mit einem vorgegebenen Bau-Budget erreichen lässt oder wie viel man investieren muss, wenn man eine bestimme Reisezeit erreichen will.

Eine Pareto-Optimierung zu berechnen, kann mathematisch sehr kompliziert sein. In der Informatik, der Mathematik und den Wirtschaftswissenschaften wird daran noch immer geforscht. Eine einfache Möglichkeit besteht darin, eine große Menge an Netzwerken zufällig zu erzeugen (also mit zufälligen Einsen und Nullen in der Verbindungsmatrix) und dann jeweils die Baukosten und die Reisezeit zu berechnen und diese in das Pareto-Diagramm einzutragen. Das ist aber nicht sehr vielversprechend, denn es gibt ja so viele Möglichkeiten, die Städte mit Kanten zu verbinden und die meisten davon sind überhaupt nicht sinnvoll.

Daher möchte ich lieber anders vorgehen, so dass ich möglichst schnell nützliche und gute Lösungen finden kann. Wenn ich bereits eine mögliche Lösung (z.B. mit minimalen Kosten oder mit minimaler Reisezeit) habe, dann kann ich diese Lösung ein bisschen verändern und dabei beobachten, in welche Richtung in dem Pareto-Diagramm ich mich bewege. Wenn die Richtung nach links oder nach unten (oder beides) geht, dann ist die neue Lösung ebenfalls gut geeignet und ich kann sie zu meiner Lösungsmenge dazu nehmen. Wenn die Richtung nach rechts oder oben geht, dann ist die neue Lösung schlechter und ich lösche sie lieber wieder. Jede neue gute Lösung kann ich dann immer weiter verändern und erzeuge so immer nur Lösungen die in der Nähe der Pareto-Kurve (also möglichst weit links unten) liegen.

An der Steigung der Pareto-Kurve oder an der Steigung der Gerade zwischen zwei Lösungen (zwei Netzwerken im Pareto-Diagramm) kann man ablesen, in welchen Verhältnis sich die Reisezeit und die Baukosten zwischen zwei verschiedenen Netzwerken verändert. Die Steigung misst also die Effizienz, wie viel Zeitersparnis pro Euro erreicht werden können. Man kann damit entscheiden, ob sich eine zusätzliche Investition lohnt oder nicht.

3. Mathematische Modellierung des Problems

Um meine Netzwerkoptimierung in Python zu programmieren, muss ich die Problemstellung in mathematischen Formeln ausrücken, die ich dann berechnen kann. Das Schienennetz wird als ein Graph beschrieben, dessen Knoten `C[i]` den Städten entsprechen und dessen Kanten `A[i][j]` den direkten Verbindungen. Ich verwende hier die Verbindungsmatrix, d.h. es ist `A[i][j] = 1`, wenn eine direkte Verbindung zwischen den Städten `C[i]` und `C[j]` existiert und sonst ist `A[i][j] = 0`.

Jede Kante hat eine Länge `L[i][j]`, die gleich der Distanz der beiden Städte `C[i]` und `C[j]` voneinander ist. Die Länge berechne ich mit den Satz des Pythagoras. Ich verwende diese Länge, um die Baukosten für eine Verbindung zu berechnen aber gleichzeitig auch um die Reisezeit zu schätzen. Beides ist, zumindest ungefähr, proportional zur Distanz. Ich vernachlässige dabei absichtlich geographische Gegebenheiten wie Gebirge oder Flüsse, wo notwendige Tunnel oder Brücken die Kosten zusätzlich beeinflussen. Die Datenerhebung dafür wäre viel aufwändiger und am Algorithmus würde sich nichts ändern, wenn die Kosten oder die Reisezeiten auf andere Art geschätzt werden.

Eine weitere wichtige Information ist, wie viele Leute von `C[i]` nach `C[j]` fahren, denn Strecken, auf denen mehr Passagiere reisen, sollten bei der Netzwerkplanung bevorzugt werden gegenüber Nebenstrecken, auf denen nur sehr wenige Passagiere reisen. Ich speichere diese Information in einer weiteren Matrix `P[i][j]`.

Ich habe beim Bundesbahn Kundenservice nachgefragt, ob sie mir sagen können wie viele Passagiere auf den verschiedenen Strecken unterwegs sind, habe aber leider keine brauchbaren Informationen erhalten. Daher musste ich die Anzahl der Passagiere schätzen. Ich bin dabei wie folgt vorgegangen.

Wenn man Leute, die dauerhaft von einer Stadt in eine andere umziehen, weglässt, kann man davon ausgehen, dass die Anzahl der Passagiere von `C[i]` nach `C[j]` genauso groß ist wie die Anzahl der Passagiere von `C[j]` nach `C[i]`, denn jeder Reisende fährt ja irgendwann wieder zurück, d.h. `P[i][j] = P[j][i]`.

Von den Menschen, die in einer Stadt `C[i]` wohnen, gehen über einen gewissen Zeitraum x% auf Reisen und der Anteil von ihnen, der eine bestimmt andere Stadt `C[j]` besucht, ist proportional zu deren Einwohnerzahl. Wenn `E[i]` die Anzahl der Einwohner der Stadt `C[i]` ist, dann ist die Gesamtbevölkerung aller Städte

$$S = \sum_i E[i]$$

Von den x% der `E[i]` Einwohner der Stadt `C[i]` reisen also

$$x\% * E[i] * \frac{E[j]}{S - E[i]}$$

nach `C[j]`. Außerdem sind

$$x\% * E[j] * \frac{E[i]}{S - E[j]}$$

der `E[j]` Einwohner von `C[j]` auf der gleichen Strecke auf der Rückreise. Insgesamt reisen also

$$P[i][j] = x\% * E[i] * E[j] * \left(\frac{1}{S - E[i]} + \frac{1}{S - E[j]}\right)$$

von C[j] nach C[j]. Der tatsächliche Prozentsatz x% ist für die Optimierung egal, wenn er in allen Städten gleich groß ist. Ich kann also z.B. einfach x = 5% annehmen.

Vor allem bei internationalen Schienennetzen hängt die Anzahl der Reisenden nicht nur von der Einwohnerzahl, sondern auch von der Distanz ab, denn es reisen mehr Passagiere im Nahverkehr (z.B. in die nächste Stadt) als im Fernverkehr (z.B. in ein anderes Land). Das habe ich berücksichtigt, indem ich die Anzahl P[i][j] der Passagiere von Stadt i nach Stadt j durch die *Distanz* L[i][j] dieser beiden Städte geteilt habe. Ich habe sogar durch das *Quadrat der Distanz* geteilt, weil dadurch der Unterschied zwischen Nahverkehr und Fernverkehr weiter verstärkt wird und der Effekt im Ergebnis deutlicher sichtbar. Damit ich die Matrix P nicht in jedem Schritt neu berechnen muss, habe ich hierbei den Luftlinien-Abstand L[i][j] benutzt und nicht die kürzesten Verbindungen S[i][j] im Netzwerk.

Die tatsächlichen Längen der Verbindungen von C[i] nach C[j] erhalte ich nämlich, wenn ich jeweils entlang der kürzesten Pfade im Netzwerk fahre. Diese Information liefert mir der Floyd-Warshall-Algorithmus, der eine Matrix S berechnet, so dass S[i][j] gleich der Länge des kürzesten Pfades von C[i] nach C[j] ist.

Mit der Verbindungsmatrix A[i][j] und den Kantenlängen L[i][j] kann ich jetzt die Baukosten K für das Schienennetz berechnen:

$$K = Baukosten\ pro\ Kilometer * \sum_{i=1}^{n} \sum_{j=1}^{n} A[i][j] * L[i][j]$$

Hierbei benutze ich den „Trick“, dass in der Summe nur die tatsächlich vorhandenen Verbindungen berücksichtigt werden (A[i][j] = 1) und die anderen Abstände zwischen nicht direkt verbundenen Städten mit 0 multipliziert werden (A[i][j] = 0) und daher nicht zur Summe beitragen.

Mit der Matrix der kürzesten Pfandlängen S[i][j] und den Passagierzahlen P[i][j] kann ich die Gesamtreisezeit Z berechnen:

$$Z = Reisezeit\ pro\ Kilometer * \sum_{i=1}^{n} \sum_{j=1}^{n} S[i][j] * P[i][j]$$

Wenn ich ein Netzwerk N1 habe und durch eine Veränderung ein anderes Netzwerk N2 daraus erzeuge (z.B. Hinzufügen oder Löschen einer Kante), dann kann ich ausrechnen, wie effizient diese Veränderung ist, d.h. um wie viel sich die Reisezeit verkürzt hat (oder verlängert hat) im Verhältnis zu den zusätzlichen (oder eingesparten) Baukosten:

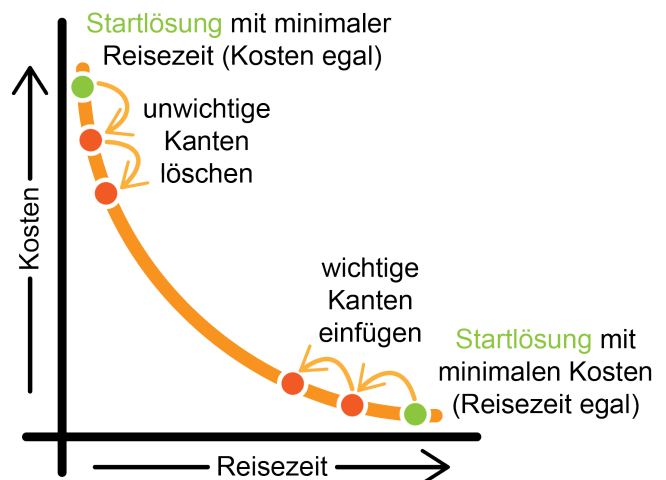
$$Effizienz = \frac{Z[N2] - Z[N1]}{K[N1] - K[N2]}$$

Die Steigung der Pareto-Kurve ist eigentlich negativ, aber ich habe die Formel so festgelegt, dass die Effizienz immer positiv ist. Das kann man sich wie folgt klar machen: Wenn ich eine Kante zum Netzwerk hinzufüge, dann steigen die Baukosten, aber die Gesamtreisezeit wird verkürzt (Zähler negativ, Nenner negativ). Umgekehrt, wenn ich eine Kante entferne, sinken die Baukosten, aber die Gesamtreisezeit erhört sich (Zähler positiv, Nenner positiv).

Weil ich im Pareto-Diagramm die Zeit auf der x-Achse und die Kosten auf der y-Achse auftrage, habe ich den folgenden Zusammenhang: Wenn die Effizienz der Kante, die ich entfernt oder hinzugefügt habe, hoch ist (wichtige Kante im Netzwerk), dann ist die Steigung der Pareto-Kurve flach (unten rechts), wenn die Effizienz der Kante gering ist (unwichtige Kante), dann ist die Steigung der Pareto-Kurve steil (oben links).

4. Optimierungsmethode

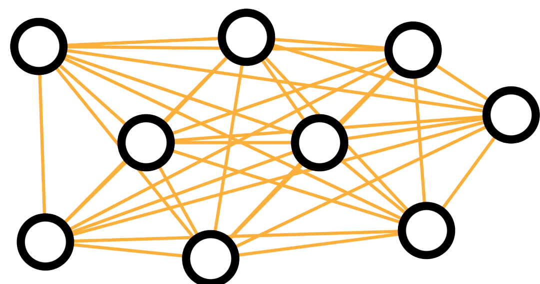
Zur Berechnung von optimalen Netzwerken, gehe ich so vor, dass ich zuerst ein Startnetzwerk berechne und dann jeweils entweder die Kante mit maximaler Effizienz hinzufüge oder die Kante mit minimaler Effizienz entferne und mich so entlang der Pareto-Kurve bewege.



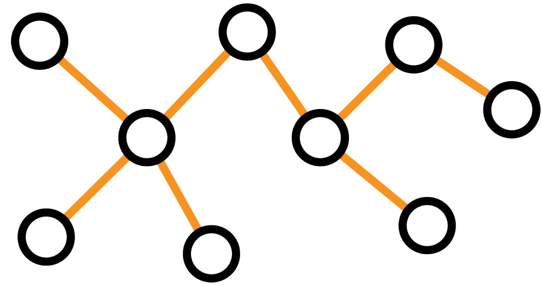
Für das Startnetzwerk habe ich verschiedene Möglichkeiten ausprobiert:

(1) <u>Vollständiger Graph</u>

Hier ist jeder Knoten mit jedem anderen Knoten direkt verbunden. Dadurch ist die Reisezeit natürlich minimal, aber die Baukosten sind extrem hoch. In meinen Experimenten habe ich z.B. ein Netzwerk zwischen den 38 größten Städten in Deutschland berechnet. Wenn jede Stadt zu jeder anderen Stadt eine direkte Verbindung hat, dann sind das (38 * 37) / 2 = 703 Verbindungen. Beim Europa-Netzwerk sind es sogar (234 * 233) / 2 = 27261 Kanten. Durch Entfernen von Kanten, können die Kosten des Netzwerks reduziert werden, aber weil es so viele Kanten sind, dauert das ziemlich lange.

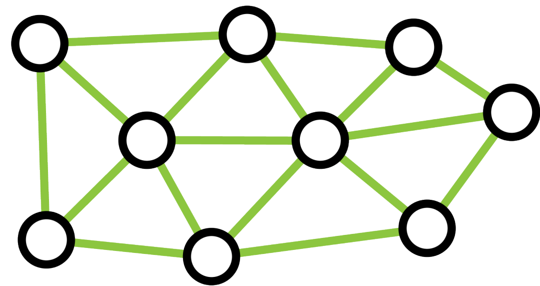


(2) Minimaler Spannbaum



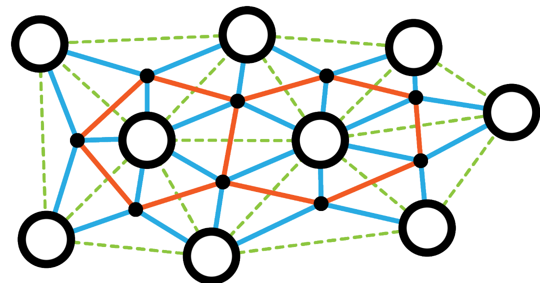
Den minimalen Spannbaum erhalte ich als Ergebnis des Kruskal-Algorithmus. Er verbindet mit geringen Baukosten alle Knoten (Städte) führt aber zu ziemlich langen Reisezeiten, da viel Umwege gefahren werden müssen. Durch das Einfügen zusätzlicher Kanten, können die Reisezeiten reduziert werden.

(3) Delaunay-Triangulierung



Die Delaunay-Triangulierung als Startlösung zu verwenden ist ein guter Kompromiss, da sie sehr viel weniger Kanten enthält als der vollständige Graph. Dadurch, dass jede Stadt mit allen ihren Nachbarn verbunden ist, sind die Reisezeiten gering, obwohl man zu entfernteren Städten manchmal mehrere Zwischenstationen hat. Manche Kanten kann man löschen und damit Baukosten sparen, ohne dass die Reisezeit zu stark ansteigt.

(4) ICE-Netzwerk



Die Delaunay-Triangulierung besteht nur aus direkten Verbindungen zwischen Städten. Dadurch gibt es keine Weichen oder Verzweigungen zwischen den Städten, was in der Realität aber schon der Fall ist. Für das ICE-Netzwerk stelle ich mir vor, dass es Hochgeschwindigkeitsverbindungen gibt, die außerhalb der Städte verlaufen (rot in der Zeichnung) und von jeder Stadt gibt es Zubringer (blau), die die Stadt an das Hochgeschwindigkeitsnetzwerk anschließen. Auch hier können wieder Kanten mit geringer Effizienz gelöscht werden. Das ICE-Netzwerk erzeuge ich, indem ich für jedes Dreieck der Delaunay-Triangulierung den Mittelpunkt einfüge und mit den drei Eckpunkten verbinde (das sind die blauen Zubringer). Dann verbinde ich die Mittelpunkte von benachbarten Dreiecken mit Hochgeschwindigkeitsgleisen (in rot).

Für die Optimierung füge ich in jedem Schritt entweder eine Kante zu meinem Netzwerk hinzu (Startnetzwerk: minimaler Spannbaum) oder ich lösche eine Kante (alle anderen

Startnetzwerke). Ich probiere alle Möglichkeiten (also jede Kante) aus und berechne, wie sich dadurch das Netzwerk verbessern oder verschlechtern würde. Dann entscheide ich mich für die Kante mit der größten Verbesserung (größte Effizienz beim Einfügen) oder der geringsten Verschlechterung (geringste Effizienz beim Löschen). Danach geht es mit den restlichen Kanten weiter.

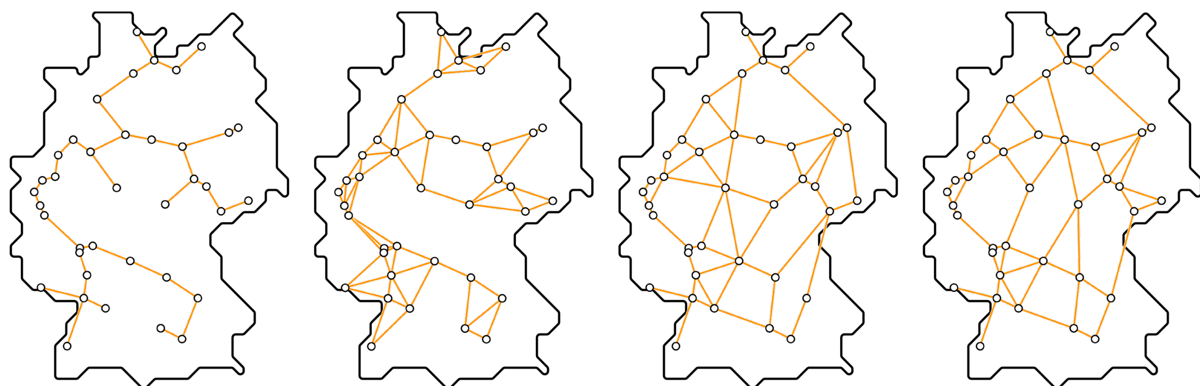
Wenn der Algorithmus einmal eine Kante ausgesucht hat, dann wird diese Entscheidung später nicht mehr geändert. Das nennt man einen „Greedy" Algorithmus [9], also einen „gierigen" Algorithmus, der nichts (keine Kante) jemals wieder hergibt. Mit einem Greedy-Algorithmus kann man zwar nicht garantieren, dass das Endergebnis wirklich die optimale Lösung ist, aber er liefert trotzdem oft sehr gute Resultate. Im Fall der Pareto-Optimierung gibt es ja sowieso nicht nur eine einzige optimale Lösung, also sollte das Greedy-Verfahren hier in der Praxis gute Ergebnisse liefern.

Der Grund warum Greedy-Algorithmen nicht immer die optimale Lösung finden, liegt daran, dass sich eine Entscheidung (welche Kante) später als nicht optimal herausstellen könnte, z.B. wenn inzwischen noch andere Kanten eingefügt wurden, die eine vorher eingefügte Kante weniger effizient machen, weil weniger kürzeste Pfade über diese Kante verlaufen. Um das herauszufinden, habe ich den Algorithmus so erweitert, dass er erst eine Reihe von Kanten nach ihrer Effizienz einfügt, dann wieder ein paar Kanten löscht, dann wieder einfügt und so weiter. Dieses Vorgehen kann tatsächlich leicht verbesserte Ergebnisse liefern, die Verbesserungen sind allerdings nicht sehr groß.

5. Ergebnisse

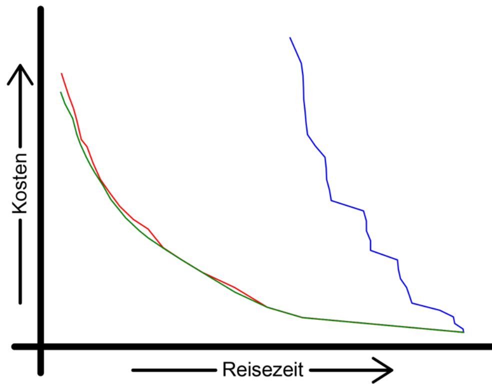
Hier zeige ich Bilder von Netzwerken und Pareto-Kurven für die verschiedenen Start-Netzwerke und die Varianten, die nur einfügen nur löschen oder beides gemischt.

Zuerst probiere ich das <u>Einfügen</u> von Kanten und vergleiche verschiedene Kriterien für die Auswahl der Kanten: (a) „billigste Kante", (b) „schnellste Reisezeit", (c) „effizienteste Kante".



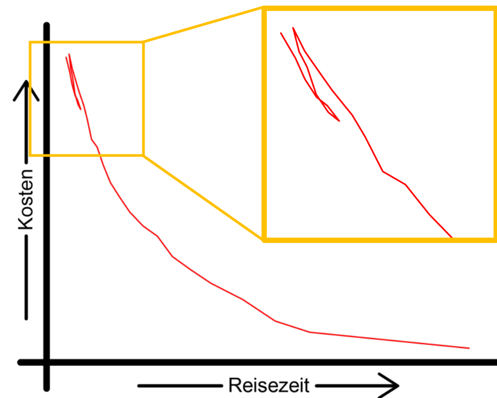
Min. Spannbaum | Kosten | Reisezeit | Effizienz

Im Pareto-Diagramm kann man die verschiedenen Kriterien vergleichen:

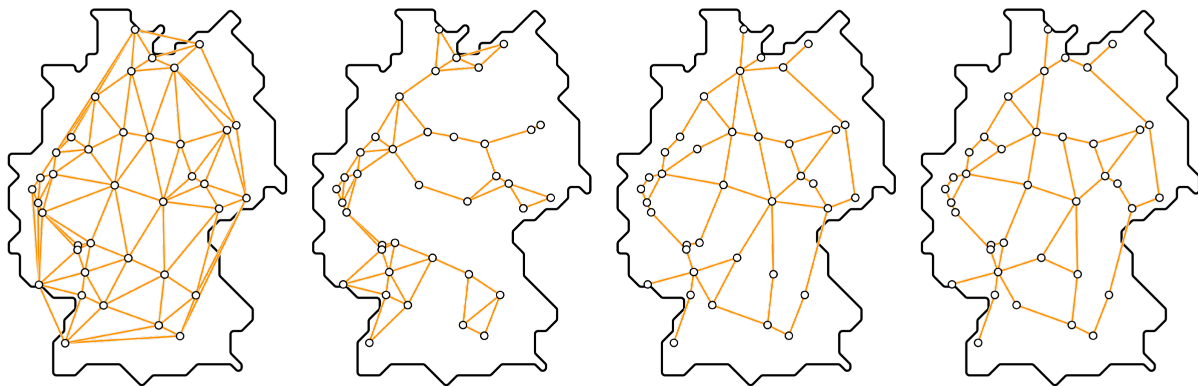


Die blaue Kurve zeigt das Einfügen der billigsten Kante, die rote Kurve das Einfügen der Kante, die die Reisezeit am meisten verkürzt und die grüne Kurve das Einfügen der effizientesten Kante (Reisezeitverringerung pro Baukosten).

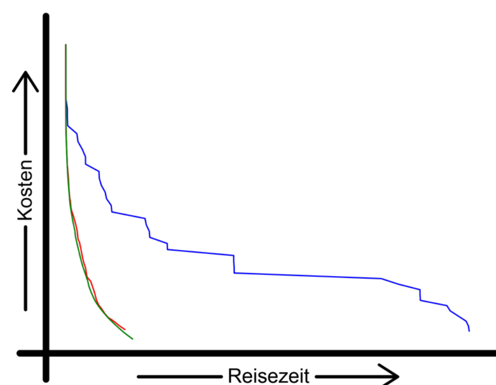
Dann probiere ich, ob sich das Ergebnis verbessern lässt, wenn ich Kanten erst einfüge, dann lösche und dann wieder einfüge. An der Pareto-Kurve (Beispiel: effizienteste Kante) sieht man, dass sich das Ergebnis zwar ein bisschen verbessert, aber der Unterschied nur sehr gering ist.

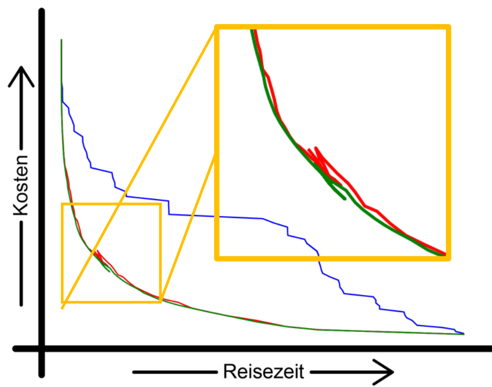


Als nächstes verwende ich die Delaunay Triangulierung als Startlösung und lösche jeweils (a) die „teuerste“ Kante, (b) die Kante mit der geringsten Reisezeitverlängerung oder (c) die Kante mit der niedrigsten Effizienz (Reisezeitverlängerung pro gesparte Baukosten).



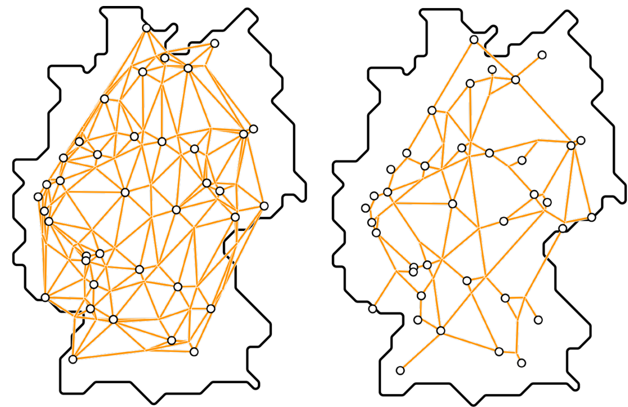
An der Pareto-Kurve sieht man wieder, dass das Effizienz-Kriterium am besten funktioniert und das Kosten-Kriterium am schlechtesten (blau: Kosten, rot: Reisezeit, grün: Effizienz).





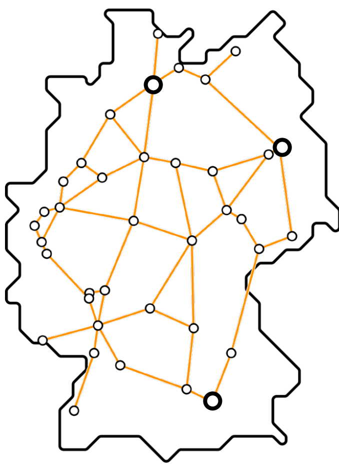
Wenn ich die Ergebnisse aus beiden Experimenten (Einfügen und Löschen) zusammen in einem Pareto-Diagramm darstelle, kann man auch gut sehen, dass die Netzwerke, die ich durch Kanteneinfügen erhalte und die Netzwerke durch Kantenlöschen zu ähnlich guten Ergebnissen führen, wobei das Kantenlöschen ein kleines bisschen besser funktioniert, aber auch etwas längere Rechenzeit für die Optimierung benötigt.

Wenn ich den vollständigen Graphen als Startlösung verwende, kommt ein Ergebnis heraus, das fast genauso aussieht, wie das bei der Delaunay Triangulierung, nur das die Rechenzeit sehr viel länger ist. Bei der Delaunay Triangulierung dauert die Berechnung auf meinem Laptop nur ein paar Minuten, beim vollständigen Graphen mehrere Stunden.

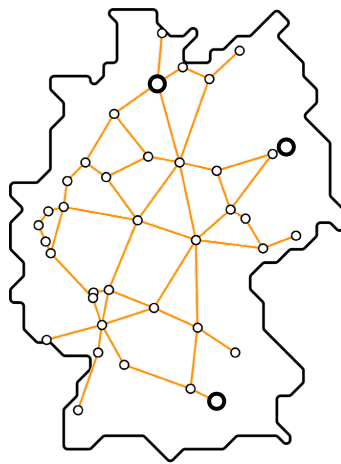


Die Experimente mit dem ICE-Netzwerk haben leider nicht so gut funktioniert, weil der Algorithmus oft Kanten gelöscht hat, die eigentlich sehr effizient aussahen und dadurch viele „Sackgassen“ entstanden sind. Ich glaube das liegt daran, dass es im ICE-Netzwerk überhaupt keine direkten Verbindungen mehr zwischen den Städten gibt und daher immer ein Umweg über zwei Zubringer gefahren werden muss. Das führt dazu, dass alle Zubringer-Kanten eine hohe Effizienz haben (ohne einen Zubringer müssen große Umwege gefahren werden) und dann der Algorithmus schwerer entscheiden kann, welche Kante gelöscht werden soll. Vielleicht wäre es sinnvoll, das ICE-Netzwerk mit der Delaunay Triangulierung zu kombinieren, aber dann hat das Startnetzwerk wieder sehr viele Kanten und die Berechnung dauert sehr lange. Das muss ich noch genauer untersuchen.

Schließlich probiere ich noch aus, welchen Einfluss die Anzahl der Reisenden pro Verbindung auf das Ergebnis hat. Ich vergleiche dafür das Netzwerk für die tatsächlichen Einwohnerzahlen der Städte (links) mit dem Netzwerk, bei dem auf allen Strecken die gleiche Zahl von Passagieren reist, d.h. alle Städte gleich groß sind (rechts):



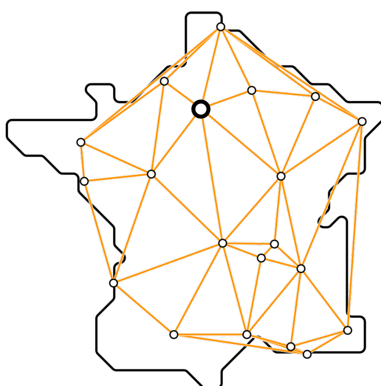
richtige Einwohnerzahlen



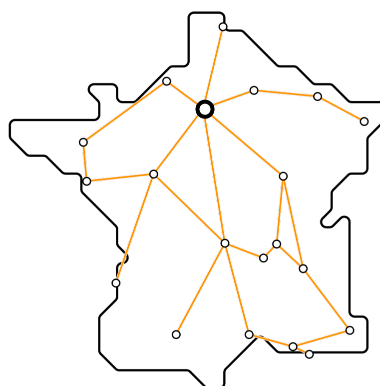
alle Städte gleich groß

Man kann deutlich sehen, dass im linken Netzwerk größerer Wert auf die Verbindungen Berlin-Hamburg und Berlin-München gelegt wird, denn in diesen Städten wohnen die meisten Menschen. Wenn alle Städte gleich groß wären, würden Berlin und München nur durch Sackgassen erreicht, weil sie am Rand des Netzwerkes liegen.

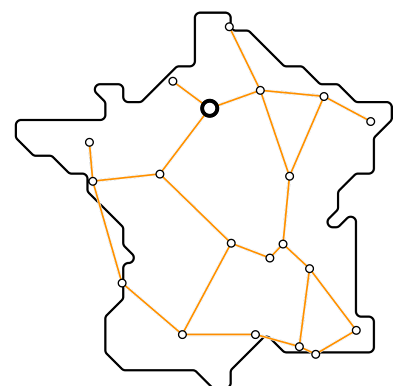
Um diesen Effekt noch deutlicher zu zeigen, habe ich meinen Algorithmus angewendet, um ein Schienen-Netz für die 20 größten Städte in Frankreich auszurechen. In Frankreich ist Paris mit großem Abstand die bevölkerungsreichste Stadt, d.h. die Verbindungen nach Paris sind viel wichtiger als andere Verbindungen.



Delaunay Triangulation

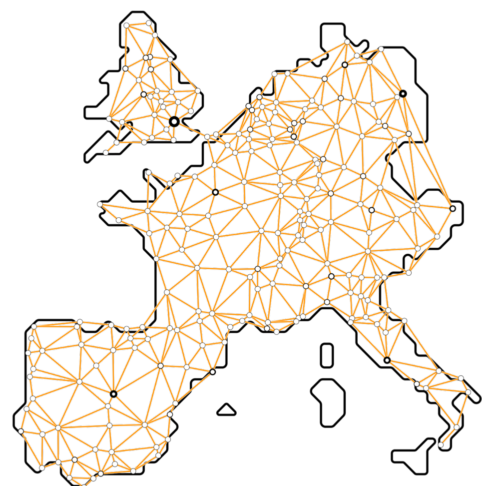


Paris die größte Stadt

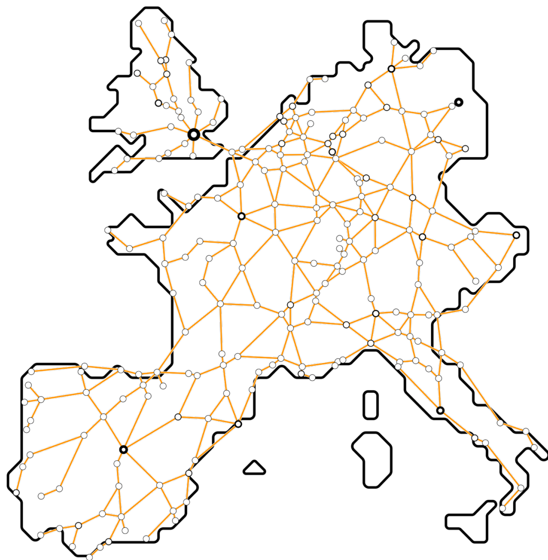


alle Städte gleich groß

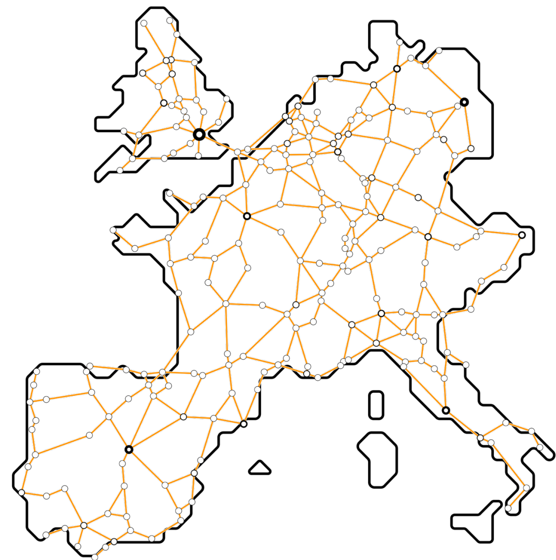
Um auch noch ein größeres Netzwerk zu berechnen, habe ich mir die Einwohnerzahlen und Koordinaten der größten 234 Städte in Westeuropa herausgesucht und habe ausgehend von der Delaunay Triangulierung (rechts, Kanten über das Meer sind entfernt außer dem Euro-Tunnel) durch Kantenlöschen Netzwerke mit maximaler Effizienz generiert. Während beim Deutschland-Netzwerk die Division der Passagierzahlen durch den Luftlinien-Abstand keinen großen Effekt hatte, zeigt sich beim Europa-Netzwerk ein großer Unterschied. Ohne die Distanz entstehen in England, Portugal und Italien



baumartige Strukturen mit großen Umwegen im Regionalverkehr. Mit den Distanzen, bleiben Regionalverbindungen in den Randgebieten erhalten.

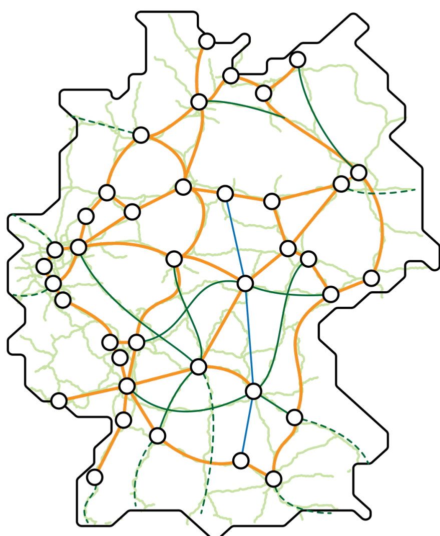


ohne Division durch Distanz
(Gleichbehandlung von Nah- und Fernverkehr)



mit Division durch Distanz zum Quadrat
(Nahverkehr wichtiger als Fernverkehr)

6. Ergebnisdiskussion, Fazit und Ausblick



Die Netzwerke, die mein Algorithmus berechnet, sehen ziemlich plausibel aus und passen sich gut der geografischen Lage der Städte und den Bevölkerungszahlen an. Am Beispiel der 38 größten deutschen Städte (links) kann man erkennen, dass die berechneten Verbindungen (orange) recht gut mit dem echten Autobahn-Netz (hellgrün) übereinstimmen. Das Autobahn-Netz verbindet natürlich mehr Städte und Regionen miteinander, die ich nicht berücksichtigt habe. Die deutlichsten Unterschiede sieht man an den Stellen, wo sich Autobahnen außerhalb von Städten kreuzen (dunkelgrün). Sowas ginge bei mir nur, wenn ich meinen Algorithmus mit dem „ICE-Netzwerk" starte, was aber bisher noch nicht so gut funktioniert hat. Das werde ich noch weiter untersuchen. Außerdem gibt es in echt z.B. eine Autobahn, die westlich (A61) und eine die östlich (A3) das Rheintal umfährt, diese geografischen Daten habe ich auch nicht verwendet. Die gestrichelten Verbindungen hat mein Algorithmus nicht erzeugt, denn das sind Verbindungen ins Ausland. Ich finde es interessant, dass mein Algorithmus sogar eine neue Verbindung: *Braunschweig – Erfurt –*

Nürnberg – Augsburg vorschlägt (blau), die es zwar in Wirklichkeit nicht gibt, die aber durchaus sinnvoll und effizient wäre.

Die Pareto-Kurven bei meinen Berechnungen zeigen sehr gut wie die Baukosten und die Reisezeiten zusammenhängen. Ich habe an den Achsen der Pareto-Diagramme absichtlich keine genauen Einheiten eingetragen. Die tatsächlichen Baukosten in Euro und Reisezeiten in Stunden würden sich ergeben, wenn ich für die „Baukosten pro Kilometer" und die „Reisezeit pro Kilometer" in den Formeln für **K** und **Z** entsprechende Werte einsetzen würde.

Der Schwerpunkt in meinem Projekt lag darauf, erstmal herauszufinden, wie man mit Netzwerken überhaupt rechnen kann und diese optimieren. Mein Algorithmus funktioniert zwar gut, er könnte aber vielleicht noch deutlich beschleunigt werden. Im Moment muss ich für *jeden* Schritt *alle* Kanten überprüfen und dafür jedes Mal den Floyd-Warshall Algorithmus ausführen. Dieser läuft mit einer dreifachen Schleife über alle Knoten. Wenn ich also die Anzahl der Knoten (Städte) in meinem Netzwerk verdopple, dann benötigt der Floyd-Warshall Algorithmus 2 * 2 * 2 = 8-mal so lange. Außerdem, da es dann doppelt so viele Kanten gibt, muss der Floyd-Warshall Algorithmus in jedem Schritt doppelt so oft ausgeführt werden und zusätzlich müssen vermutlich auch doppelt so viele Kanten aus der Delaunay Triangulierung entfernt werden. Insgesamt erhöht sich die Rechenzeit des Algorithmus also ungefähr um den Faktor 2 * 2 * 8 = 32, wenn sich die Anzahl der Knoten verdoppelt. Daran erkennt man, dass die Rechenzeit sehr schnell von Minuten zu Stunden oder sogar Tagen ansteigen kann, wenn ich größere Netzwerke berechnen will. Für mein Europa-Netzwerk hat mein Laptop über eine Woche lang gerechnet.

Die Rechenzeit könnte ich deutlich beschleunigen, wenn ich den Floyd-Warshall Algorithmus nicht jedes Mal komplett neu ausführen müsste, nur weil eine einzelne Kante gelöscht oder hinzugefügt wurde. Viele der kürzesten Pfade ändern sich vermutlich gar nicht wegen dieser einen Kante. Beim Einfügen einer Kante ist das tatsächlich möglich, beim Löschen einer Kante haben Forscher aber bisher noch keine schnelle Lösung gefunden.

Eine andere Möglichkeit, die Berechnung zu beschleunigen, wäre es, nicht immer alle Kanten in jedem Schritt zu überprüfen. Vielleicht könnte ich irgendwie vorhersagen, welche Kante sich lohnt auszuprobieren. Oder ich könnte in jedem Schritt gleich mehrere Kanten einfügen oder löschen, wenn ich mehrere Kanten finde, die eine ähnliche (hohe oder niedrige) Effizienz haben.

7. Quellen und Literaturverzeichnis

[1] https://db-watch.de/kostenexplosion-bei-der-generalsanierung-der-riedbahn/

[2] „Algorithmen kapieren“, Aditya Bhargava

[3] „Algorithmen und Datenstrukturen für Dummies“, A. Gogol-Döring, T. Letschert

[4] https://de.wikipedia.org/wiki/Algorithmus_von_Kruskal

[5] https://de.wikipedia.org/wiki/Algorithmus_von_Floyd_und_Warshall

[6] https://de.wikipedia.org/wiki/Delaunay-Triangulierung

[7] https://de.wikipedia.org/wiki/Voronoi-Diagramm

[8] https://de.wikipedia.org/wiki/Mehrzieloptimierung

[9] https://de.wikipedia.org/wiki/Greedy-Algorithmus

[10] „Algorithmen in Python“, David Kopec