Heuristiken zur Erstellung von Linienkonzepten

20. August 2008

vorgelegt von

Rasmus Fuhse

aus

Friedrichshafen

Betreuer: Prof. Anita Schöbel

angefertigt
im Institut für numerische Mathematik
der Georg-August Universität zu Göttingen

Erklärung

Ich,
versichere hiermit, dass ich die vorliegende Diplomarbeit mit dem Titel: „Heuristiken zur Erstellung von Linienkonzepten“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Göttingen, den 20. August 2008 Rasmus Fuhse

Inhaltsverzeichnis

1 Einführung

Mache aus großen Problemen kleine und aus kleinen Problemen gar keine.
Angeblich Chinesisches Sprichwort
(entnommen einer Postkarte)

Tagtäglich fahren tausende Busse und Bahnen durch die Gegend und transportieren Menschen von Ort zu Ort - ebenso wie ungezählt viele Flugzeuge, und sogar einige Schiffe und Hovercrafts. Das tun sie nun schon seit so einiger Zeit, aber erst seit relativ kurzer Zeit gibt es das Bestreben, diese Verkehrsnetze mathematisch zu betrachten. Die Frage, die sich stellt, ist recht einfach: kann man da nicht noch so einiges besser machen?
Vermutlich hat sich jeder, der einmal Opfer der allgemeinen Verkehrsführung geworden ist, der einmal zu viel umsteigen musste, der ganz planmäßig mit hundert anderen Menschen viel zu lange auf einen Anschlusszug warten musste oder der auch nur ansehen musste, wie ein Anschlusszug vor seiner Nase weggefahren ist, so etwas mal gefragt.
Dabei liegt das Interesse, hier etwas zu verbessern, sowohl bei den Fahrgästen, die (hoffentlich) komfortabler reisen können, als auch bei den Betreiberunternehmen, die Kosten sparen können.
Bisher sind diese Netze und Regeln, nach denen sich dort draußen so viel bewegt, mehr oder weniger mit bloßem Menschenverstand erstellt worden. Nichts gegen den guten alten Menschenverstand, aber sobald die Probleme etwas größer werden, reicht er oftmals nicht aus. Und Verkehrsnetze sind in den über hundert Jahren seit ihrer Entstehung (je nach Definition sind sie sogar noch älter) zu einem sehr großen Problem geworden - zumindest aus mathematischer Sicht.
Als Beispiel mag hier D. Shanno dienen (Quelle: [2] Seite 12), einer der ersten Mathematiker, der Anfang der 1990er aktiv ein Verkehrsnetz optimieren sollte und zwar für die Fluggesellschaft Delta Airlines. Er schaffte es, das Problem, alle Fluglinien und Flugwege zu optimieren, in eine 45000 x 101000 Matrix zu transformieren, und hat und mit dem inneren Punkte Verfahren eine Lösung heraus bekommen. Diese Lösung hätte Delta Airlines 6000000 US-Dollar eingespart - pro Woche.
Endgültig umgesetzt wurde dieser Flugplan nicht, weil er zu viele fundamentale Umwälzungen in der Betriebsstruktur erfordert hätte. Aber das Problem zeigt recht spektakulär auf, welchen Einfluss die Mathematik auf große Probleme haben kann, die bisher stets mit Menschenverstand (und bestimmt nicht von dummen Menschen oder ohne intensiven Nachdenkens) gelöst worden sind.
Das Problem mit diesen Problemen ist leider immer noch, dass sie zu kompliziert sind. Man muss immer wissen, was man eigentlich optimieren will: Fahrzeit, Kosten, Umsteigevorgänge, Umsteigezeiten, aber auch weiterführende Probleme wie Regeln zum Umgang mit verspäteten Zügen. Selbst Standorte von neuen Bahnhöfen oder Betriebsstellen kann man mathematisch ermitteln, wenn man ein günstiges Modell verwendet.
Dabei muss man stets auf viele Sachen achten: es dürfen keine zwei Züge gleichzeitig auf einem Gleis stehen, es müssen alle Fahrgäste ihr Ziel erreichen, Fahrzeuge müssen alle tausend Kilometer gewartet werden, es dürfen keine exorbitanten Kosten entstehen und und und.
Selbstverständlich hat Herr Shanno von Delta Airlines nicht alle diese Punkte in seine Matrix einfließen lassen. Wollte man das tun, würde man vor einer unlösbaren Aufgabe stehen. Statt dessen tut man einfach das, was jenes angeblich chinesische Sprichwort uns rät: man zerteilt das Problem in viele kleine Probleme.
Viele dieser kleinen Probleme sind schon gelöst und es gibt Algorithmen, die sowohl in der Theorie als auch in der Praxis gute Ergebnisse liefern.

Aber wie so häufig, wenn man etwas zerteilt hat, muss man es auch wieder zusammenfügen. Deswegen gründete Frau Professor Schöbel an der Universität Göttingen die LinTim Arbeitsgruppe, was für Lineplanning und Timetabling steht. Ziel dieser Arbeitsgruppe ist es, mehrere Algorithmen, die sich bewährt haben, zusammenzufügen, sodass man sie auf einem zentralen Datenformat aufbauend beliebig hintereinander ausführen kann. So kann ein Algorithmus zuerst einen Fahrplan optimieren und der nächste kann dann mit einem optimierten Fahrplan ein Verspätungsmanagement (also die Frage, ob ein Zug auf einen verspäteten Zug warten soll) optimieren. Auf diese Weise kann man allen Schritten der Verkehrsplanung einmal hautnah zusehen und vor allem auch sehen, ob die Algorithmen auch gut zusammen arbeiten. Aus vielen kleinen Lösungen wird dann wieder eine große gewonnen, was natürlich für das große Problem nur eine heuristische also ungenaue Herangehensweise ist, aber das lässt sich nunmal nicht ändern.

Ziel dieser Arbeit, die Sie gerade in den Händen halten, ist es, eine Lücke der Verkehrsplanung zu schließen: und zwar ist das die Linienplanung. Einíge Algorithmen sind in der Lage, einen Linienverbund (also beispielsweise mehrere Buslinien oder Zuglinien) zu optimieren. Aber die Frage, die bisher unbeantwortet ist, lautet: wie bekommt man eigentlich diesen ersten Linienverbund? Dabei kann schon an dieser Stelle eine Menge falsch gemacht werden, was wir versuchen können, mit ein wenig Mathematik richtig zu machen.
Auch obwohl der Linienverbund, den wir in den folgenden Kapiteln erarbeiten wollen, später noch von anderen Algorithmen eventuell optimiert wird, so müssen wir uns trotzdem im Klaren darüber sein, dass diese späteren Algorithmen nur kleine Fehler ausbügeln können, denn sie sind oftmals nur Heuristiken oder behandeln stark eingeschränkte Probleme. Wenn sie alle Fehler ausbügeln würden, würde es sich hierbei schließlich nicht um eine Lücke in der Verkehrsplanung handeln.
Wir sind also in einer einzigartigen Lage, gerade am Anfang der Verkehrsplanung gewissermaßen die Weichen für ein gut funktionierendes Verkehrsnetz zu legen. Alles, was wir falsch machen, kann bis zum Schluss falsch bleiben, aber anders herum wird alles, was wir richtig machen, bis zum Schluss richtig bleiben. Eine enorme Verantwortung, der wir selbstverständlich gerne entgegen treten.

2 Verkehrsplanung und Liniennetze

Zuerst einmal wollen wir unser Problem in ein mathematisches Modell wandeln, mit dem wir dann vernünftig arbeiten können. Der Einfachheit des Sprachgebrauchs wegen stellen wir uns vor, wir optimieren ein Netz von Bussen, aber genauso könnten wir ein Bahnnetz oder Flugnetze betrachten. Vielleicht aber sind Busnetze auch für unsere Vorstellung der Materie etwas dienlicher.
Im Folgenden wollen wir aus einem real existierenden Busnetz ein mathematisches Konstrukt werden lassen. Dabei wird es einige Aspekte des realen Netzes geben, die wir mit unserem Modell nicht nachbilden können. Einige Aspekte lassen wir schlicht weg, weil es zu kompliziert wäre, sich darüber Gedanken zu machen. Andere Aspekte könnten wir gar nicht behandeln, selbst wenn wir wollten, weil uns die praktischen Anwendungsdaten fehlen würden, um damit arbeiten zu können. Das Modell soll einfach sein, denn unser Problem wird noch schnell genug sehr schwierig werden.

2.1 Begriffe und Notationen

Definition: 2.1.1 *Ein PTN (public transportation network) ist ein endlicher und ungerichteter Graph $G = (V, E)$ mit Knoten V und Kanten E. Dabei repräsentieren die Knoten die Haltepunkte bzw. Stationen und die Kanten die direkten Verbindungen zwischen den Knoten, wie sie in der Form von Straßen oder Schienen vorkommen. Zudem ist der Graph gewichtet, wobei die Gewichte t_e für alle $e \in E$ die Reisedauer eines Gefährts betragen, die es entlang der Kante e benötigt. Alle Elemente aus E haben die Form $\{v_i, v_j\}$, wobei $v_i \neq v_j$ Knoten des PTN sind.*

Man könnte den Graphen auch als gerichtet betrachten, was die Größe des Problems in unseren Algorithmen verdoppeln würde, weil ja doppelt so viele Kanten im PTN betrachtet werden müssen. In diesem ungerichteten Netzwerk hingegen ist es unerheblich, ob ein Passagier von Punkt A nach Punkt B oder aber von B nach A möchte - er wird dieselben Fahrtstrecken und dieselben Linien benutzen und die dieselbe Fahrtzeit benötigen. Das macht uns viele Berechnungen einfacher und schneller.
In einem gerichteten Graphen verändert sich allerdings auch unser Problem. Die Linien, die wir suchen, hätten nicht länger die Form von Linien mit einem Anfangs- und einem Endpunkt, sondern müssten im Grunde Kreise sein, damit ein Bus am Ende wieder da steht, wo er los gefahren ist. Diese Kreise können natürlich gravierend anderes aussehen als unsere althergebrachte Vorstellung von Buslinien. Gerade beim Auftreten von Einbahnstraßen würden diese Kreise zwangsläufig entstehen. Die Umstellung auf gerichtete Graphen hätte daher zum Teil gravierende Folgen für die Algorithmen, die wir später anführen werden. Der Fall für gerichtete Graphen müsste also komplett eigenständig behandelt werden.

Definition: 2.1.2 *Sei ein PTN gegeben. Ein Pfad im PTN ist ein Vektor von Kanten $p = (e_1, ..., e_k)$, sodass für alle $1 \leq i < k$ die Kanten e_i und e_{i+1} inzidieren.*
Eine Linie in dem PTN ist ein Pfad l $(l_1, ..., l_z)$ bzw. sein Inverses also $(l_z, ..., l_1)$ zusammen mit einer natürlichen Zahl f, die die Frequenz der Linie darstellt und anzeigt, wie häufig in einem bestimmten Zeitintervall die Linie fährt.
Eine Menge von endlich vielen Linien nennen wir entweder Linienpool, Linienverbund oder Linienkonzept. Ein Linienpool bezeichnet vor allem eine übergroße Menge von lose zusammenhängenden Linien, ein Linienverbund eine gut zusammen passende Menge von bezahlbar vielen Linien und ein Linienkonzept ist ein Linienverbund mit ausgeklügelten Frequenzen.

Die Unterscheidung in Linienpool, Linienverbund und Linienkonzept ist natürlich weniger mathematisch sondern eher sprachlich bedingt.

Weil wir in der Linienplanung die Frequenzen nicht gebrauchen werden (Frequenzen sind erst für die Fahrplanerstellung wichtig), seien für die folgenden Seiten die Frequenzen stets 1.
Weil unser PTN ungerichtet ist, ist es für uns unerheblich, ob eine Linie bei a anfängt und bei b aufhört oder umgekehrt, sofern der Weg dazwischen gleich ist. Deswegen identifitzieren wir die Linie sowohl mit dem Pfad als auch mit dem Inversen des Pfades.
Wir haben jetzt also ein Netzwerk und können darin Linien definieren. Als Letztes brauchen wir jetzt nur noch Passagiere. Für uns ist wichtig, wo ein Passagier einsteigt und wo der Passagier aussteigen will.

Definition: 2.1.3 *Sei G ein PTN. Eine Matrix $W \in Mat(V \times V, \mathbb{N}_0)$, deren Einträge w_{ij} angeben, wieviele Passagiere in G von Haltestelle i zur Haltestelle j reisen wollen, nennen wir die Origin-Destination Matrix oder kurz OD-Matrix.*

Damit hat man schon einmal ein sehr handliches mathematisches Modell für die Linienplanung. Mehr werden wir vorerst nicht brauchen. Unser Ziel wird es sein, einen Algorithmus zu finden, der als Input ein PTN hat und als Output einen wie auch immer *guten* Linienpool fördert.

3 Netzerstellung - vom Bedarf zum Liniennetz

Nun haben wir die Grundlagen gelegt und wissen, was ein PTN ist und welche Linien wir darin suchen. Als nächstes wollen wir uns dem eigentlichen Problem zuwenden.
Die bisherigen gängigen Algorithmen der Linienplanung verlangen als Input einen bestehenden Linienpool bzw. Linienverbund. Die zentrale Frage dieses Dokumentes hier ist nun, wie wir schnell und effizient an eine solche Menge von Linien heran kommen, wenn wir nicht von Anfang an eine haben?
Aber wo ist da die Schwierigkeit? Prinzipiell ist es doch recht einfach, Linien zu konstruieren, die sich lange durch das Netzwerk schlengeln. Und wenn man mehrere dieser Linien zusammenbringt, hat man schon einen Linienpool, mit dem die anderen Algorithmen arbeiten können. Solche Pools würden sich enorm schnell herstellen lassen - ohne viel komplizierte Mathematik dahinter.
Das Problem ist hier aber, dass so eine Menge von Linien an sich auch eine gewisse Güte für uns hat. Sobald mehrere Linien zusammen gefasst sind, kann man als Fahrgast sich fragen, ob diese paar Linien gut zusammen passen oder nicht. Ein schlechter Linienverbund wäre es auf jeden Fall, wenn der *durchschnittliche* Fahrgast häufig umsteigen muss und/oder sehr lange Fahrtzeiten bis zu seinem Ziel hat, weil die Busse ständig Umwege fahren. In jedem Fall ist ein Linienverbund für den Fahrgast umso besser, je größer er ist, weil es für den Gast mehr Möglichkeiten gibt, sich fortzubewegen. Und ein Linienverbund mit allen möglichen Linien ist für jeden denkbaren Fahrgast optimal, weil er über mindestens eine Linie direkt, ohne Umsteigen und ohne Umweg zu seinem Ziel kommt. So viele Linien sind aber teuer und aufwändig. Da kommt auch unser sprachlicher Unterschied zwischen Linien-Pool und -Verbund ins Spiel. Die hohe Kunst soll es daher sein, einen möglichst kleinen Linienverbund zu finden, der bezahlbar ist und die Fahrgäste schnell und mit möglichst wenig Umsteigen zum Ziel bringt.

3.1 Zielfunktion

Das Hauptanliegen soll es sein, dass bei dem entstandenen Liniennetz die Passagiere möglichst wenig umsteigen müssen. Zugleich sollen natürlich auch alle Passagiere an ihr Ziel kommen. Für dieses Problem gibt es zwei triviale (aber dennoch höchst unpraktische) Lösungen:

1. Von jeder Haltestelle zu jeder anderen fährt ein Bus. Dazu ist unser Linienpool gigantisch groß. Das hat den Vorteil, dass jeder Passagier sehr schnell an seinem Bestimmungsort ist und nie umsteigen muss. Es führt allerdings auch dazu, dass die Busse wenig belegt sind, zudem kostet das Netz den Betreiber zuviel Geld und überhaupt würde die ganze Stadt nur noch von Bussen befahren werden.

2. Durch die ganze Stadt fährt nur ein Bus, in den jeder Passagier einsteigt. Dieser eine Bus fährt nacheinander alle Haltestellen der Stadt ab. Folge ist, dass niemand umsteigen muss, aber dafür verlängert sich die Fahrzeit für die meisten Passagiere ungemein.

Als Kompromiss wollen wir gerne, dass eine moderate Zahl an Bussen durch die Stadt fährt, aber zugleich für jeden Passagier sowohl die Umsteigezahl als auch die absolute Reisedauer möglichst gering bleibt.
Sofort kommt einem in den Sinn, um die erste Situation abzuwenden, sollte man sich ein maximales Budget setzen, das nicht überschritten werden darf. Um die zweite Situation in den Griff zu bekommen, muss unsere Zielfunktion eine Summe aus Umsteigezahlen der Passagiere und der Fahrzeiten derselben sein. Etwa so:

$$t(p, L) = r \cdot Umsteigezahl(p, L) + Fahrzeit(p, L)$$

Dabei ist p unser Fahrgast, wie er in der OD-Matrix auftaucht, L der gegebene Linienverbund und r eine Konstante, die jeder Passagier sich selbst wählt und angibt, ab wievielen Minuten mehr der Fahrzeit er lieber einmal umsteigen würde. Beispielsweise würde er umsteigen, wenn er 5 Minuten Fahrzeit einsparen könnte - dann würde $r = 5$ gelten. Der Umsteigevorgang selbst benötigt natürlich auch Zeit. Der Einfachheit halber gehen wir von einem statischen r aus, das der Durchschnitt jedes Fahrgastes ist und die Umsteigezeit sowie einen Unkomfortabilitätsfaktor zugleich darstellt.
Und unser Problem soll dann sein:

$$\min_{L \in \mathcal{L}_G} \sum_{p \in P} t(p, L)$$

Wobei $\mathcal{L}_G$ die Potenzmenge aller möglichen Linien also die Menge aller Mengen von Linien in unserem PTN und P die Menge aller charakteristischen Fahrgäste ist, die in der OD-Matrix berücksichtig worden sind. Das bedeutet auch, dass wir bedarfsgesteuert arbeiten wollen, also über alle Passagiere summiert wird und nicht über alle Paare von Haltestellen. Dadurch werden hochfrequentierte Fahrwünsche eher berücksichtigt.

3.2 Die Nebenbedingungen

Vorhin haben wir schon festgestellt, dass es eine sinnvolle Nebenbedingung ist, wenn man die Anzahl der Busse limitiert.

1. Verhindert das, dass als Ergebnis unserer Optimierung $n(n-1)$ Buslinien (nämlich eine für jede mögliche Verknüpfung zweier Knoten im PTN) durch die Stadt fahren.

2. Hält es vermutlich auch die Kosten des Betreiberunternehmens und damit die Kosten für eine Fahrkarte auf einem gesunden Niveau.

Wir wollen die Kosten des Linienpools L folgendermaßen berechnen:

$$c(L) = \sum_{l \in L} (Startkosten + |l| \cdot Folgekosten)$$

Dabei ist $|l|$ die Länge der Linie l im PTN. Startkosten und Folgekosten sind positive Konstanten.

3.3 Problemstellungen

Im Folgenden wollen wir unsere Forderungen an das Endprodukt konkretisieren und mehrere unterschiedliche Probleme definieren, auf die später immer wieder verwiesen wird.

- **Definition: 3.3.1** *Paralleles Linienproblem mit Budget: Gegeben sei ein PTN mit Graph G und den Passagieren P sowie einer Kostenfunktion $c : \mathcal{L}_G \to \mathbb{R}$ und einem maximalen Budget $b \in \mathbb{R}$. Gesucht ist*

$$\min_{L \in \mathcal{L}_G} \left\{ \sum_{p \in P} t(p) \;\middle|\; c(L) \leq b \right\}$$

mit zugehörigem Linienverbund $\hat{L}$, den wir einen optimalen Linienverbund nennen.

Dies ist das Problem, das wir oben beschrieben haben. Wenn t und c lineare Abbildungen wären (was insbesondere t in unserem Fall nicht ist), so hätten wir ein ganzzahliges Problem vor uns, was in seiner Allgemeinheit vermutlich schon NP-Vollständig wäre. Mit nichtlinearen t und c haben wir also ein sehr schweres Problem vor uns.

- **Definition: 3.3.2** *Paralleles Linienproblem ohne Budget: Gegeben sei ein PTN mit Graph G und den Passagieren P. Gesucht ist*

$$\min_{L \in \mathcal{L}_G} \left\{ \sum_{p \in P} t(p) \right\}$$

mit zugehörigem Linienverbund $\hat{L}$, den wir einen optimalen Linienverbund nennen.

Dieses Problem ist leicht zu lösen. Weil es kein Budget gibt, können wir einfach ALLE Linien, die denkbar sind, in den Linienpool einfügen und haben den besten Linienpool. Dieses Problem werden wir nicht behandeln, weil es trivial lösbar und zu teuer für die Praxis ist.

- **Definition: 3.3.3** *Serielles Linienproblem ohne Budget: Gegeben sei ein PTN mit Graph G und den Passagieren P. Gesucht ist*

$$\min_{L \in \mathcal{L}_G} \left\{ \sum_{p \in P} t(p) \;\middle|\; \forall e \in E \; \exists! \; l \in L \text{ mit } e \in l \right\}$$

mit zugehörigem Linienverbund $\hat{L}$, den wir einen optimalen Linienverbund nennen.

Dieses Problem ist stark eingeschränkt dadurch, dass keine zwei Linien im Verbund L auf derselben Kante liegen dürfen. Warum soll das so sein? Wir werden noch sehen, dass es ein komplizierter Zusatz ist, Linien auch parallel laufen zu lassen. Das würde unser Problem mathematisch unübersichtlich machen. Für dieses serielle Problem können wir dagegen bessere mathematische Aussagen für unsere Algorithmen finden.

- **Definition: 3.3.4** *Serielles Linienproblem mit Budget: Gegeben sei ein PTN mit Graph G und den Passagieren P sowie einer Kostenfunktion $c : \mathcal{L}_G \to \mathbb{R}$ und einem maximalen Budget $b \in \mathbb{R}$. Gesucht ist*

$$\min_{L \in \mathcal{L}_G} \left\{ \sum_{p \in P} t(p) \;\middle|\; c(L) \leq b \;\wedge\; \forall e \in E \; \exists! \; l \in L \text{ mit } e \in l \right\}$$

mit zugehörigem Linienverbund $\hat{L}$, den wir einen optimalen Linienverbund nennen.

Dies ist ein schweres Problem, weil die Kostenfunktion c nicht allein linear von der Länge aller Linien abhängen muss. Aber für uns ist es auch nicht so interessant.

Für den eifrigen Leser gibt es im Anhang eine Tabelle, die uns zeigt, welche Probleme wir im Folgenden mit welchen Algorithmen zu lösen versuchen werden:

In den folgenden Seiten und Kapiteln werden wir uns hauptsächlich um das parallele Linienproblem mit Budget kümmern und - gewissermaßen als Vorbereitung darauf - auch mit dem seriellen Linienproblem ohne Budget.

4 Erste Heuristiken

Da wir schon alle Definitionen haben und uns das Problem mittlerweile auch gut bekannt sein dürfte, fangen wir nun damit an, das Problem zu lösen. Zuerst wollen wir uns dem Seriellen Linienproblem ohne Budget zuwenden. Es ist zwar nicht genau das, was wir am Ende haben wollen, aber es ist wesentlich übersichtlicher und wir können mathematisch sinnvolle Aussagen finden.

4.1 Kürzeste Wege in Netzwerken

Wir müssen wissen, welche Strecken von den Passagieren am meisten befahren werden, und ebenso, welcher Passagier nach einem Streckenabschnitt in welchen anderen Streckenabschnitt will.
Kürzeste Wege lassen sich mit dem Algorithmus von Dijkstra in quadratischer Laufzeit finden. In nichtnegativen Graphen (also mit strikt positiven Kantengewichten) kann dieser einen Baum aller kürzesten Wege erzeugen von einem beliebigen Knoten x ausgehend.

Wir brauchen diese kürzesten Wege, um zu wissen, welche Wege unsere Passagiere bzw. OD-Paare durch das PTN nehmen wollen, um an ihr Ziel zu kommen. Dieses Wissen nutzen wir, um lokal an jeder Haltestelle zu bestimmen, welche Linie dort am besten in welche Richtung weiter fahren sollte, indem wir die Zahlen der Passagiere, die an jener Haltestelle an- und wieder abfahren, geschickt ausnutzen.

4.2 Matchingprobleme

Man könnte sich vorstellen, dass unser Linienverbund schon ziemlich gut wäre, wenn summiert über alle Haltestellen möglichst wenige Passagiere umsteigen also ihre Linie wechseln müssen. Dies wird die Idee unseres Algorithmus. Um dies zu erreichen, betrachten wir jede Haltestelle getrennt und wollen wissen, welche Passagiere auf welcher Linie ankommen und wo sie hin wollen. Anschließend definieren wir daraus die Linien neu, sodass weniger Passagiere umsteigen müssen. Wir bedienen uns dabei bekannter Matching-Algorithmen.

Definition: 4.2.1 *Sei $G = (V, E)$ ein ungerichteter Graph. Dann ist $M \subset E$ ein Matching, wenn keine zwei Kanten aus M inzidieren also den selben Knoten berühren.*
Ein Matching heißt perfekt, wenn es alle Knoten des Graphen überdeckt.
Mit überdecken eines Knotens meinen wir hier, dass der Knoten von mindestens einer Kante des Matchings berührt wird.
Das Gewicht eines Matchings ist $\sum_{e \in M} t_e$.

Wir wollen für ein PTN G einen Linienverbund L assoziieren mit n Matchings für n Matchingprobleme. Das bedeutet, aus den Matchings können wir eine Menge von Linien ableiten und aus dem Linienverbund wiederum das Matching, die Beziehung ist also eineindeutig.
Zuerst einmal, wie wir uns das vorzustellen haben:

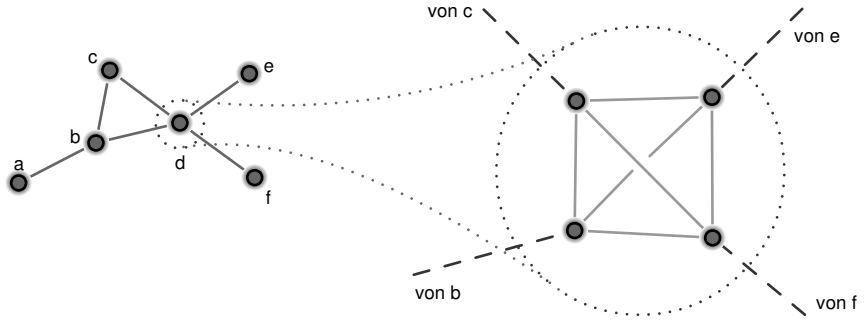
Definition: 4.2.2 *Sei ein PTN $G = (V, E)$ gegeben. Wir bezeichnen einen Graphen $M_i = (X_i, Y_i)$ als einen Kanten-Matchinggraphen zu einem Knoten i, wenn gilt:*

- *$|X_i|$ ist der Knotengrad von $i \in V$, also die Anzahl der zu i adjazenten Kanten,*
- *M_i ist ein vollständiger Graph,*

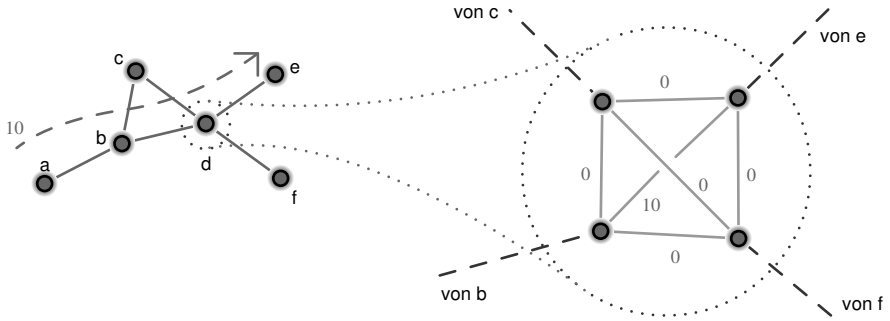
- *M_i hat nichtnegative natürliche Kantengewichte.*

Wir bezeichnen die Knoten von M_i auch mit den gleichen Namen wie die zu i adjazenten Kanten, die sie repräsentieren, damit die Relation zwischen Kante in G und Knoten in M_i erhalten bleibt.

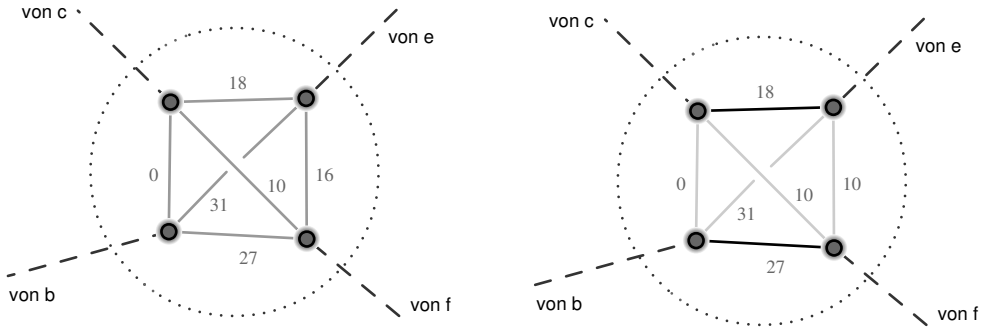
Wir betrachten nun eine Haltestelle unseres PTN als ein Matchingproblem, wobei wir die eingehenden Kanten als Knoten unseres Matchinggraphen sehen:



Der entstehende Matchinggraph soll vollständig sein, also jeder Knoten mit jedem anderen verbunden sein.
Jetzt betrachten wir ein OD-Paar, das sich auf einem kürzesten Weg durch den Graphen bewegt. Nehmen wir beispielsweise 10 Personen, die von Knoten a nach Knoten e wollen:



Dieses OD-Paar wird entsprechend ihrer Anzahl auf der Kante eingetragen, über der sie reisen würde - in diesem Fall also von Richtung b nach Richtung e.
Wenn wir das mit allen OD-Paaren machen, bekommt der Matchinggraph ganz ordentliche Gewichte. Jetzt können wir uns auf die Suche nach einem gewichtsmaximalen Matching zu diesem neu gewonnen Graphen begeben.



Haben wir das gefunden, können wir dieses Matching mit Linien assoziieren, indem

wir sagen, dass die Linien gerade auf den Kanten des gewichtsmaximalen Matchings verlaufen. Das können wir machen, weil durch die Beschaffenheit eines Matchings gewährleistet ist, dass keine zwei Linien auf einer Kante verlaufen. Damit passt der neue Linienverbund zu unserem seriellen Linienproblem.
Bringen wir in dieses Konzept jetzt etwas Formalität:

Definition: 4.2.3 *Seien ein PTN $G = (V, E)$, ein Linienverbund L und die zu G passenden Kanten-Matchingprobleme $M_1 = (X_1, Y_1), ..., M_n = (X_n, Y_n)$ gegeben. Wir definieren die Abbildungen*

$$S_1 : \mathcal{L}_G \quad \rightarrow \begin{pmatrix} \mathcal{M}(Y_1) \\ \vdots \\ \mathcal{M}(Y_n) \end{pmatrix}$$

$$L \quad \mapsto \begin{pmatrix} \left\{ \{e_j, e_k\} \in M_1 \text{ wenn } \exists l \in L \text{ mit } l(a) = e_j \text{ und } l(a+1) = e_k \right\} \\ \vdots \\ \left\{ \{e_j, e_k\} \in M_n \text{ wenn } \exists l \in L \text{ mit } l(a) = e_j \text{ und } l(a+1) = e_k \right\} \end{pmatrix}$$

und

$$S_2 : \begin{pmatrix} \mathcal{M}(Y_1) \\ \vdots \\ \mathcal{M}(Y_n) \end{pmatrix} \rightarrow \mathcal{L}_G$$

$$\begin{pmatrix} m_1 \\ \vdots \\ m_n \end{pmatrix} \mapsto \left\{ (e_{i_1}, ..., e_{i_x}) \;\middle|\; \begin{array}{c} \forall 1 \leq j < x \;\; \exists e' \in \bigcup_{k=1}^{n} m_k \;\; \text{mit} \;\; e' = \{e_{i_j}, e_{i_{j+1}}\} \\ \text{und} \\ \left(x > 1 \text{ oder } \nexists e' \in \bigcup_{k=1}^{n} m_k \;\; \text{mit} \;\; e' = \{e_{i_j}, e_{i_{j+1}}\}\right) \end{array} \right\}$$

wobei $\mathcal{M}(Y_i)$ Menge aller Matchings der Kantenmenge Y_i ist und $l(a)$ die a-te Kante der Linie l.

Satz: 4.2.4 *Seien ein PTN $G = (V, E)$ und ein Linienverbund L als Lösung zum seriellen Linienproblem ohne Budget gegeben. Wir können den Linienverbund L eindeutig identifizieren mit Matchings $m_1, ..., m_n$ zu den Matchingproblemen $M_1, ..., M_n$ oder anders ausgedrückt: S_1 und S_2 sind bijektiv und es gilt $S_2 \circ S_1 = id$.*

Beweis: Zuerst zeigen wir: S_1 ist wohldefiniert und injektiv.

- Für die Wohldefiniertheit nehmen wir an, dass es einen Linienverbund L gebe, sodass eine der Komponenten von $S_1(L)$ kein Matching ist. Dann würde es in dieser Komponente m_k zwei Kanten e_i und e_j geben, die einen gemeinsamen Knoten haben. Daraus folgt, dass es laut Vorschrift zwei Linien in L gibt, für die gilt $l_1(a) = l_2(b)$ für bestimmte a und b. Mit anderen Worten, würden zwei Linien aus L über ein- und dieselbe Kante in G fahren, was aber nicht erlaubt ist, da L eine gültige Lösung des seriellen Linienproblems ist ($\lightning$). Also ist S_1 wohldefiniert.
- Injektivität: Gegeben seien zwei Linienverbünde L_1 und L_2 mit $S_1(L_1) = S_1(L_2) = (m_1, ..., m_n)$. Für jede Kante $\{e_i, e_j\}$ in $\bigcup_{k=1}^{n} m_k$ gibt es sowohl in L_1 als auch in L_2 eine Linie, die e_i und direkt danach e_j überfährt. Weil beide Linienverbünde das serielle Linienproblem lösen, gibt es zudem auch

keine andere Linie (sowohl in L_1 als auch L_2), die irgendeine dieser Kanten befährt. Also müssen diese beiden Linien von L_1 bzw. L_2 insgesamt gleich sein (bis auf Inversion, also Vertauschung der Anfangs- und Endpunkte), weil diese Eigenschaft auch für alle anderen Kanten in $\bigcup_{k=1}^{n} m_k$ gilt, und die Linien gewissermaßen die gleichen Richtungen einnehmen. Da dies für alle Linien gilt, gilt also $L_1 = L_2$ bis auf Inversion, was für uns unerheblich ist.

Jetzt zeigen wir das Gleiche noch für S_2:

- Wohldefiniertheit: Zuerst muss gezeigt werden, dass S_2 auch wirklich eine Lösung des seriellen Linienproblems erzeugt aus den Matchings. $S_2((m_1, ..., m_n))$ ist eine Menge von Vektoren mit Kanten als Komponenten. Jeweils zwei aufeinanderfolgende Kanten inzidieren, weil es eine Kante in einem Matching gibt, sodass beide Kanten Element der Matchingkante sind. Demnach sind die Vektoren von Kanten auch gültige Pfade und damit $S_2((m_1, ..., m_n))$ ein Verbund von Linien. Weil $m_1, ..., m_n$ Matchings sind, taucht eine Kante aus G in höchstens einer Linie auf. Dass auch tatsächlich jede Kante in G vom Linienverbund abgedeckt wird, ist, zumindest wenn sie in keiner Matchingkante auftaucht, nicht so klar, wird aber durch die zusätzliche Eigenschaft

$$(x > 1 \ oder \ \nexists e' \in \bigcup_{k=1}^{n} m_k \ \ mit \ \ e' = \{e_{i_j}, e_{i_{j+1}}\})$$

von S_2 sichergestellt. Hier werden alle Linien mit einer Kante in den Linienverbund hinzufügt, deren eine Kante von keiner Matchingkante gematcht wird. Diese zusätzliche Eigenschaft klärt, dass auch genau die einkantigen Linien hinzugenommen werden, die nötig sind. Sind die n Matchings beispielsweise alle leere Matchings, so sind alle Linien im Verbund einkantig. L ist also ein Linienverbund, der das serielle Linienproblem löst. Damit ist also auch die Wohldefiniertheit gezeigt.

- Injektivität: Seien zwei Vektoren von Matchings gegeben mit $S_2((m_1, ..., m_n)) = S_2((m'_1, ..., m'_n)) = L$. Wir betrachten in L zuerst alle Linien, die nur über eine Kante fahren. Diese Kanten haben zwangsweise in $\bigcup_{k=1}^{n} m_k$ und $\bigcup_{k=1}^{n} m'_k$ keine Matchingkante, die die Linie matcht. Also sind die ungematchten Knoten in beiden Vektoren der Matchings gleich. Unterschiedlich kann also nicht sein, was von $(m_1, ..., m_n)$ bzw. $(m'_1, ..., m'_n)$ gematcht ist, sondern nur, wie es gematcht ist.
  Für eine längere Linie $(e_{i_1}, ..., e_{i_x})$ in L gilt $\forall 1 \leq j < x \ \exists e' \in \bigcup_{k=1}^{n} m_k \ mit \ e' = \{e_{i_j}, e_{i_{j+1}}\}$. Dieses e' muss aber in beiden Vektoren $(m_1, ..., m_n)$ und $(m'_1, ..., m'_n)$ in der gleichen Komponente auftauchen, weil die Kanten e_{i_j} und $e_{i_{j+1}}$ inzidieren müssen, der Graph aber einfach ist, es also keine Schlaufen und Doppelkanten geben kann. Dann aber sind alle Matchings gleich, also auch $(m_1, ..., m_n) = (m'_1, ..., m'_n)$. Daraus folgt, dass S_2 injektiv ist.

Nach dem Auswahlaxiom folgt jetzt, dass S_1 und S_2 ebenfalls surjektiv und damit bijektiv sind. Bleibt nur noch zu zeigen, dass auch wirklich $S_2 \circ S_1 = id$. Das liegt gerade daran, dass aus einer Kantenreihenfolge (x, y) von einem $l \in L$ eine Matchingkante $\{x, y\}$ in einem Matching wird.
Diese Matchingkante wird durch S_2 wieder den Kanten zugeordnet, also x und y. Es fährt wegen der Wohldefiniertheit von S_2 am Ende genau eine Linie über x und eine über y und es ist gewährleistet, dass dies ein- und dieselbe Linie l ist, die insbesondere x und y direkt hintereinander abfährt. Also bleibt die Kantenreihenfolge jeder Linie beibehalten, was wieder den gleichen Linienverbund ergibt. Damit gilt also auch $S_2 \circ S_1 = id$ und natürlich auch $S_1 \circ S_2 = id$. □

Lemma: 4.2.5 *Die Anzahl aller Lösungen des seriellen Linienproblems sind gleich die Anzahl aller Vektoren* $(m_1, ..., m_n)$ *mit* $m_i \in \mathcal{M}(Y_i)$ $\forall i = 1, ..., n$, *wobei* $M_i = (X_i, Y_i)$.

Beweis: Dies folgt daraus, dass S_1 eine Bijektion ist, also zwischen gleichmächtigen Mengen abbildet. Da die Menge aller Matchings in einem endlichen Graphen endlich ist, ist also auch die Menge aller Lösungen des seriellen Linienproblems endlich und wegen der Gleichmächtig gerade auch gleich der Anzahl der Vektoren aller Matchings. □

Jetzt zeigen wir noch einen Zusammenhang zwischen den Linien und den einzelnen Matchings in Bezug auf die Güte eines Linienverbundes. Mit anderen Worten: man kann die Güte eines Verbundes zu einem guten Teil an den Matchings ablesen.

Lemma: 4.2.6 *Seien ein PTN* $G = (V, E)$ *und ein Linienverbund L gegeben. Für jeden Fahrgast* $p \in P$ *ermitteln wir einen (nicht alle) kürzesten Weg. Betrachten wir ein zu einer Haltestelle i zugehöriges Matchingproblem* M_i *und seien die Kantengewichte dieses Matchingproblems gerade*

$$t_e = |P_i| \text{ mit } P_i = \{p \in P \mid \text{kürzester Weg von } p \text{ führt über } i\}$$

für alle $e \in M_i$. *Dann gibt es für die Fahrgäste im Linienverbund L genau so viele Umsteigevorgänge, wie das aufsummierte Gewicht aller Kanten beträgt, die nicht Element einer Komponente von* $S_1(L) = (m_1, ..., m_n)$ *sind.*

Beweis: Jeder Fahrgast, der als Zahl auf einer ungematchten Kante hinzugefügt wurde durch seinen kürzesten Weg, muss an der Haltestelle einmal umsteigen, weil es keine Linie gibt, die den selben Weg nimmt wie der Fahrgast. Auf der anderen Seite, muss kein Fahrgast, der als Zahl auf einer gematchten Kante hinzugefügt wurde, an der Haltestelle umsteigen, weil in L eine Linie existiert, die den Weg des Fahrgastes durchfährt. Also müssen an jeder Haltestelle genau so viele Personen umsteigen, wie das Gewicht aller Kanten beträgt, die nicht im Matching sind. □

Im Folgenden wollen wir ein Matching im Vektor $S_1(L)$ ersetzen durch ein gewichtsmaximales Matching.
Ein gewichtsmaximales Matching in einem allgemeinen Graphen lässt sich mittels des Algorithmusses von Edmonds-Karp in polynomialer Zeit finden (vergleiche [4] Seite 227). Das ist nicht gerade gut, lässt sich aber vertreten, wenn man bedenkt, dass die Matchingprobleme in der Praxis wesentlich kleiner ausfallen als die Größe des PTNs, weil nicht die Haltestellen, sondern die Anzahl der durch die Haltestelle verlaufenden Linien für die Größe der Matchingprobleme verantwortlich ist.

4.3 Subalgorithmus 1

Mit den kürzesten Wegen und den Matchingproblemen können wir nun einen kleinen Algorithmus entwerfen, der immerhin lokal einen bestehenden Linienverbund optimieren kann.

Input: PTN $G = (V, E)$ mit OD-Matrix W und einem Linienverbund L auf G, einem spezifischen Knoten $v \in V$ und zu jedem Fahrgast p einen für ihn günstigsten Pfad, in dem über das PTN reisen möchte.

1. Betrachte Kanten-Matchingprobleme $M_1, ..., M_n$ und erzeuge Matchings $m_1, ..., m_n$ durch $S_1(L)$;
2. trage anhand der kürzesten Wege der Fahrgäste die Kantengewichte in M_v auf;
3. berechne m_v für M_v neu als gewichtsmaximales Matching;
4. $L' = S_2((m_1, ..., m_v, ..., m_n))$.

Output: Linienverbund L'.

Damit können wir einen Linienverbund lokal an einer Haltestelle optimieren, indem wir die Umsteigezahlen minimieren.

Satz: 4.3.1 *Sei $G = (V, E)$ ein PTN mit der OD-Matrix W und zudem ein Knoten $v \in V$ gegeben. Der Subalgorithmus 1 verschlechtert einen Zielfunktionswert bezüglich des seriellen Linienproblems ohne Budget nicht.*

Beweis: Bevor wir den Subalgorithmus 1 einmal anwenden, teilen wir unsere Fahrgastmenge P auf in Fahrgäste, deren ermittelter kürzester Weg über Knoten v verläuft, die wir P_v nennen. Wir behalten aber auch im Kopf, dass diese kürzesten Wege nicht eindeutig sein müssen. Von jedem Fahrgast zählt nur ein kürzester Weg und ob ein eventuell anderer gleichlanger Weg über Knoten v laufen würde, können wir jetzt nicht wissen.

$$\sum_{p \in P} t(p, L) = \sum_{p \in P_v} t(p, L) + \sum_{p \in P \setminus P_v} t(p, L)$$

Jetzt wenden wir den Subalgorithmus 1 an. Naturgemäß bilden alle $p \in P_v$ gerade die Kantengewichte des Matchingproblems M_v.
Nehmen wir nun an, dass der Subalgorithmus 1 den Zielfunktionswert verschlechtert hat. Die zurückgelegten Wege der Fahrgäste können sich durch den Algorithmus nicht verschlechtert haben, weil wir das PTN nicht verändert haben. Daher müssen sich die Umsteigezahlen der Fahrgäste verändert haben. Weil aber nach Lemma 4.2.6 die Umsteigezahlen gleich des Gewichts aller ungematchten Kanten in den Matchingproblemen sind, und wir dieses gerade im Subalgorithmus 1 minimiert haben, kann es nicht schlechter sein als zuvor. ↯
Damit hat sich der Zielfunktionswert des seriellen Linienproblems entweder verbessert oder ist gleich geblieben. □

Die positive Aussage dieses Satzes ist natürlich, dass wir den Subalgorithmus so häufig anwenden können, wie wir wollen, und an beliebigen Knoten des PTN, sodass der Zielfunktionswert sich entweder verbessert oder gleich bleibt. Wenn wir das einfach frech tun, bekommen wir eine erste Heuristik:

4.4 Heuristik H_0

Sei ein serielles Linienproblem ohne Budget gegeben, also ein PTN G und eine OD-Matrix W.

1. $L = \varnothing$; $\forall e \in E$ füge Weg (e) zu L hinzu.
2. Initialisiere die n Matchingprobleme $M_1, ..., M_n$.
3. Berechne zu jedem Fahrgast seinen bevorzugten Reisepfad im PTN.

4. Für alle Knoten $v_i \in V$ führe Subalgorithmus 1 an v_i aus.

5. Output: Linienverbund L.

Lemma: 4.4.1 *Sei $G = (V, E)$ ein PTN mit der OD-Matrix W. Dann kann sich der Zielfunktionswert bezüglich des seriellen Linienproblems ohne Budget nie nach Durchführung des H_0 Algorithmus verschlechtern.*

Beweis: Es wird in diesem Algorithmus H_0 genau $n = |V|$ mal der Subalgorithmus 1 angewendet. Die anderen Schritte verändern den Linienverbund nicht. Also kann sich nach Satz 4.3.1 der Zielfunktionswert nicht verschlechtern. □

Satz: 4.4.2 *Sei $G = (V, E)$ ein PTN und ein kreisfreier Graph und sei die OD-Matrix W gegeben. Dann liefert H_0 eine Optimallösung für das serielle Linienproblem ohne Budget.*

Beweis: Zuerst einmal nutzen wir die Kreisfreiheit von G aus um festzustellen, dass die Wege in G eindeutig sind - damit natürlich auch die kürzesten Wege. Also sind auch die Gewichte der Matchingprobleme $M_1, ..., M_n$ eindeutig bestimmt - auch unabhängig von jeglichen Linien, die später noch gefunden werden würden.
In Schritt 2 gehen wir davon aus, dass unsere Matching-Algorithmen an sich korrekt sind und eine Optimallösungen liefern. Sofern die Optimallösungen von $M_1, ..., M_n$ alle eindeutig sind, ist auch unser Endergebnis eindeutig.
Jetzt nehmen wir noch an, dass es eine bessere Lösung für das Linienproblem geben würde. Die Wege der Passagiere in G müssten trotzdem gleich sein. Demnach ist die Fahrzeit ohne Umsteigebestrafung für jeden Passagier identisch. Was bleibt, ist also, dass in unserer besseren Lösung weniger Passagiere umsteigen müssen als in der von H_0 gefundenen Lösung. Also müssten die Matching-Algorithmen nicht die Optimallösung gefunden haben, was aber ein Widerspruch zur Korrektheit der Algorithmen wäre. ↯
Demnach liefert H_0 eine Optimallösung für das Linienkonzeptproblem, die auch eindeutig ist, sofern die Matchings $M_1, ..., M_n$ eindeutige Lösungen besitzen. □

Für Graphen mit Kreisen können wir keine derartige Optimalitätsaussage treffen, denn es ist nicht sicher, dass die Passagiere nach dem Erstellen des Linienverbunds auch wirklich die Wege befahren, die die kürzesten im PTN sind. Die Fahrgäste werden am Ende einen Weg wählen, wo ihre persönliche Zielfunktion minimal wird. Das bedeutet, dass sie eventuell einen kleinen Umweg in Kauf nehmen, wenn sie dafür weniger umsteigen müssen. Ein Beispiel hierfür wird nach dem nächsten Algorithmus, der nur eine Erweiterung von H_0 ist, angegeben.
Um das berechnen zu können, brauchen wir noch ein rechnerisches Instrument, mit dem man den für die Fahrgäste kürzesten Weg bezüglich ihrer Zielfunktion $t(p) = r \cdot Umsteigezahl(p) + Fahrzeit(p)$ ermittelt. Danach können wir unseren Algorithmus H_0 weiter entwickeln.

5 Dynamische Heuristik

Zuerst stellen wir erst einmal fest, dass unser bisheriges Netzwerk gar nicht zulässt, dass ein Passagier mit einer Buslinie fährt - bisher bewegt er sich immer nur auf dem ursprünglichen Graphen G und versucht, dort den kürzesten Weg für sich zu finden.
Wir werden ein neues Netzwerk brauchen, bei dem der Passagier an jeder Haltestelle die Möglichkeit hat, seine Linie zu wechseln.

5.1 Change&Go Graphen

Unser neues Netzwerk nennen wir einen Change&Go Graphen. Der C&G Graph ist wesentlich größer als der ursprüngliche Graph G, weil jede Haltestelle mehrmals vorkommen kann.

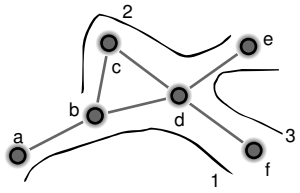
Definition: 5.1.1 *Sei $G = (V, E)$ ein PTN und $L = \{l_1, ..., l_p\}$ ein Linienpool dazu und r eine natürliche Zahl, die die Umsteigezeit der Passagiere zwischen zwei Linien bedeutet. Der Change&Go Graph G' zu G und L besteht aus:*

- *Alle Knoten $v \in V$, die wir die Ursprungsknoten nennen.*
- *Für jede Linie $l_p = (v_1, ..., v_{|l_p|})$ alle Knoten $v_1, ..., v_{|l_p|}$ einmal mit den neuen Namen $v_{1\_p\_x}, ..., v_{|l_p|\_p\_x}$, wobei sich das x ergibt als wie häufig die Linie p schon an der Haltestelle gewesen ist.*
- *Für jede Linie l_p werden alle Kanten entlang von l_p für $v_{1\_p\_x}, ..., v_{|l_p|\_p\_x}$ übernommen und die Gewichte beibehalten.*
- *Für alle $i = 1, ..., n$ werden alle Knoten der Form $v_{i\_p\_x}$ mit Kanten vom Gewicht $\frac{r}{2}$ mit dem Knoten i verbunden.*

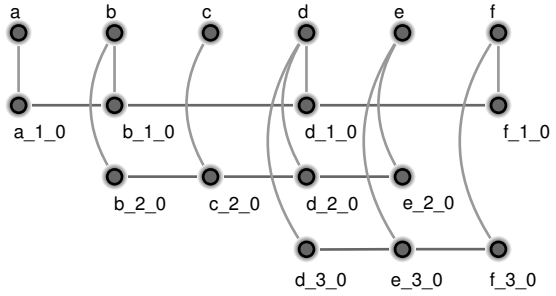
Um die Beziehung zwischen G und G' nicht zu verlieren, statten wir G' mit den Funktionen

- *halt(x) gleich des ersten Eintrags a des Nicht-Ursprungsknoten $x = a\_b\_c$ von G' oder entsprechend gleich x, wenn x ein Ursprungsknoten ist.*
- *linie(x) = linie(a_b_c) gleich dem zweiten Eintrag b von $x = a\_b\_c$ bzw. 0, wenn x ein Ursprungsknoten in G' ist.*

aus.



Graph G mit Linien 1, 2 und 3

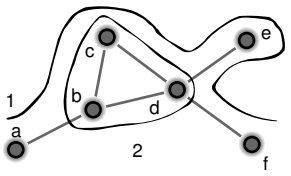


Change & Go Graph G' von G

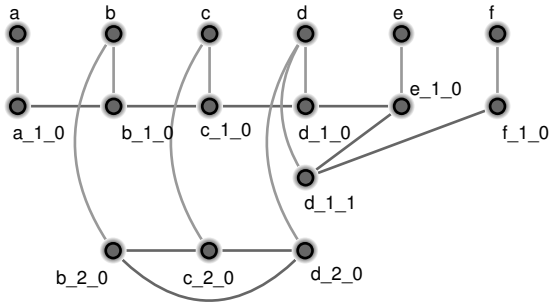
Wir haben im Grunde die Passagiere veranlasst, sich nicht länger über die klassischen Kanten aus E von Haltestelle zu Haltestelle zu bewegen, sondern direkt über die Linien. Dabei fahren die Linien alle Knoten dieser Linie an, und wenn ein Passagier umsteigen möchte, so bewegt er sich vom Knoten d_1_0 (nur um ein Beispiel

zu nennen) zum Knoten d und dann von d noch einmal zu d_2_0. Pro Umsteigen sind also zwei Kanten für den Passagier zu überqueren, wobei die Kantengewichte $\frac{r}{2}$ sind, womit wir wieder bei der eigentlich Umsteigebestrafung r sind.
Für die Reisedauer eines Passagiers von einem Knoten a zu einem anderen Knoten b werden jetzt allerdings im Change&Go Graphen r mehr Minuten benötigt als vorher, weil jede Reise mit einer halben Umsteigezeit beginnt und auch endet. Das ist aber nicht schlimm, denn diese Umsteigezeit ist identisch für alle Passagiere und kann leicht wieder subtrahiert werden.

Der Begriff des Change&Go Graphen stammt von Dr. Susanne Weißmann geb. Scholl ([6]). Der hier verwendete Graph ist allerdings an zwei Stellen leicht abgeändert worden. Zum einen sah der ursprüngliche Change&Go Graph vor, dass an einer Haltestelle jeder Knoten v_i_x mit jedem anderen Knoten v_j_y verbunden wird, also einen vollständigen Subgraphen bildet. Das wurde ersetzt durch weniger Kanten und einem zusätzlichen Knoten v, der dafür eine eindeutige Ein-/Ausstiegsadresse der Fahrgäste ist.
Und die zweite Änderung ist der zusätzliche Index x, der zuvor nicht berücksichtigt wurde. In den meisten Fällen ist das auch nicht schlimm, ihn nicht zu berücksichtigen. Aber sobald eine Linie eine Haltestelle anfährt, die sie schon zuvor angefahren hat, besteht die Möglichkeit, dass ein Fahrgast von der Linie in die Linie selbst umsteigen kann. Dadurch, dass dies hier mit mehreren Knoten dargestellt wird, kann dieser Umsteigevorgang ganz normal bestraft werden. In gewisser Weise erlauben wir uns damit einfach auch Linien, die im Kreis fahren und Haltestellen mehrfach anfahren.



Graph G mit Linien 1 und 2 *mit Kreis*



Change & Go Graph G' von G

Man bemerke auch, dass zwischen allen Linien, die an einer Haltestelle halten, die selbe Umsteigezeit existiert. In einem detaillierteren Modell könnte es Sinn ergeben, die Frequenz von ankommender und abfahrender Linie in die Umsteigezeit mit einzuberechnen. Aber weil wir noch keine anderen Frequenzen als 1 haben, nützt uns dieses Vorhaben noch nichts. Aber beispielsweise bei der Fahrplanerstellung oder der Berechnung der Frequenzen kann es sinnvoll sein, Umsteigezeiten in Abhängigkeit der Frequenzen zu betrachten. Auch dafür ließe sich ein abgewandelter Change&Go Graph verwenden.

Lemma: 5.1.2 *Sei ein PTN G und ein Linienverbund L gegeben. Die Transformation von G und L in den Change&Go Graphen G' nennen wir T_G. So ist T_G bijektiv.*

Beweis: Um Bijektivität einer Abbildung zu zeigen, müssen wir Wohldefiniertheit, Injektivität und Surjektivität zeigen:

- Wohldefiniertheit: Hier haben wir nicht viel zu zeigen. Laut unserer Definition können wir aus jedem Linienverbund auf G einen Change&Go Graphen gewinnen. Wie oben am Beispiel gezeigt wurde, können im Linienverbund sogar Linien mit Kreisen und nicht-einfache Linien auftauchen. Ein leerer Linienverbund würde auf einen Graphen mit nur den Ursprungsknoten abgebildet werden, was allerdings immer noch ein Change&Go Graph ist. Also ist unser Bild immer ein Change&Go Graph.

- Injektivität: Sei ein zweiter Linienverbund L' gegeben, sodass gilt $T_G(L) = T_G(L') = G'$. Angenommen, es gebe in L mehr Linien als in L', dann würde $T_G(L)$ anders aussehen als $T_G(L')$, weil es in in $T_G(L)$ einen Knoten der Form $v_{a\_|L|\_x}$ geben würde, der für $T_G(L')$ unmöglich wäre. Ebenso kann L' nicht mehr Linien als L haben, also gilt auf jeden Fall $|L| = |L'|$.
  Angenommen, es gibt in L eine Linie $l_i \notin L'$. l_i unterscheidet sich also in mehr als nur Inversion (also dem Vertauschen der Reihenfolge) von jeder anderen Linie in L'. Wenn sie sich unterscheidet in der Anzahl der Kanten in G, die sie befährt, dann folgt daraus, dass $T_G(L)$ in der Linie, also allen Knoten der Form $v_{a\_l_i\_x}$, größer oder kleiner als $T_G(L')$ ist, was aber nicht sein kann, weil dies beides ja gleich sein soll. Also kann diese Linie sich nur noch in der Reihenfolge zu allen Linien von L' unterscheiden. Dann aber hätten $T_G(L)$ und $T_G(L')$ unterschiedliche Kanten, was ebenfalls nicht sein kann. $\lightning$. Dann aber muss $L = L'$ gelten. Damit ist T_G injektiv.

- Surjektivität: Das ist ebenfals nicht schwer zu beweisen, weil Punkte 2,3 und 4 des Change&Go Graphen direkt von Linien abhängen und Punkt 1 die Ursprungsknoten sind, die von G abhängen. Oder anders ausgedrückt: die Surjektivität gilt erst, weil wir Change&Go Graphen als die Bildmenge von T_G definiert haben.

□

Definition: 5.1.3 *Von einem Umsteigevorgang in einem Change&Go Graphen G' sprechen wir, wenn ein Pfad $p = (p_1, ..., p_s)$ derart ist, dass p_x und p_{x+2} jeweils keine Ursprungsknoten sind und unterschiedlichen Linienindex aufweisen.*

Lemma: 5.1.4 *Sei G ein PTN und L ein Linienverbund. Dann entspricht die Länge $length(p)$ des kürzesten Weges eines Fahrgastes im Change&Go-Graphen G' gleich dem Zielfunktionswert $t(p, L)+r = r \cdot Umsteigezahl(p, L)+Fahrzeit(p, L)+r$. Also:*

$$length(p) - r = t(p, L) = r \cdot Umsteigezahl(p, L) + Fahrzeit(p, L) + r$$

Beweis: Dies ist leicht zu sehen. Der kürzeste Weg im Change&Go Graphen ist der gleiche wie ein Weg unter der Zielfunktion, weil im C&G Graphen Umsteigevorgänge genau so bestraft werden wie in der Zielfunktion, nämlich mit r. Die Bewegungsmöglichkeiten zwischen den Haltestellen ist im C&G Graphen genau so wie im PTN, weil die Kantengewichte übernommen werden. Also muss man beim Weg des Fahrgastes im C&G Graphen nur die Ein- und Aussteigezeit von jeweils $\frac{r}{2}$ abziehen, sodass beides gleich ist. □

Wir wollen nun in G' kürzeste Wege zwischen den Ursprungsknoten herausfinden und anschließend in die Matchingprobleme als Kantengewichte auftragen, wie wir es beim H_0 Algorithmus auch schon getan haben.

Definition: 5.1.5 *Sei G ein PTN, L ein Linienverbund und G' der Change&Go Graph zu G und L. Sei P die Menge aller Fahrgäste. Für alle $p \in P$ ermittle den kürzesten Weg in G', path(p).*

1. *Streiche alle Knoten aus path(p), die Ursprungsknoten von G' sind.*
2. *Für alle Paare $path(p)_i$ und $path(p)_{i+2}$ erhöhe in Matchingproblem $M_{halt(path(p)_{i+1})}$ Gewicht der Kante $\{\{halt(path(p)_i), halt(path(p)_{i+1})\}, \{halt(path(p)_{i+1}), halt(path(p)_{i+2})\}$ um eins.*

Die so entstandenen Gewichte nennen wir die Kundengewichte der Matchingprobleme.

Diesen Vorgang nennen wir Ermitteln der Kantengewichte von $M_1, ..., M_n$ aus G', wenn wir nur die Wege erhalten wollen.

Lemma: 5.1.6 *Subalgorithmus 1 verschlechtert sich auch mit den kürzesten Wegen in G' in keinem Schritt bezüglich des Zielfunktionswertes des seriellen Linienproblems.*

Beweis: Wir können aus den kürzesten Wegen in G' die Wege der Fahrgäste in G rekonstruieren. Weil dies die tatsächlichen Wege der Fahrgäste sind, gilt also auch immer noch der Beweis von Satz 4.3.1. □

Lemma: 5.1.7 *Sei $G = (V, E)$ ein PTN und L ein Linienverbund auf G, bei dem keine Linie ein Kreis ist. Sei nun $G' = (V', E')$ der Change&Go Graph von G mit dem Verbund L, so gilt, dass $|V'| \leq (|L| + 1)|V|$ und $|E'| \leq |V||L| + (|L| - 1)|V|$.*

Beweis: Der Beweis ist recht simpel und konstruktiv. Jede Linie kann maximal alle $|V|$ Knoten beinhalten, weil jede Linie kreisfrei sein soll. Das sind dann $|L||V|$ Knoten. Zuletzt bleiben die ursprünglichen Knoten $v \in V$ erhalten, was also maximal $|L||V| + |V| = (|L| + 1)|V| = |V'|$ Knoten ergibt.
Als zweites müssen wir noch die Kantenzahl nach oben abschätzen: Wir haben $|L||V|$ Knoten, denen alle eine Linie zugeordnet ist. Weil die Linien kreisfrei sein sollen, hat jeder dieser $|L||V|$ Knoten höchstens zwei Nachbarn innerhalb der Linie, was $|L| - 1$ Kanten pro Linie macht. Das sind dann insgesamt $(|L| - 1)|V|$ „Linienkanten“. Hinzu kommen die Umsteigekanten an jeder Haltestelle. Da es pro Haltestelle höchstens $|L|$ Knoten gibt, die alle mit dem einen Hauptknoten der Haltestelle über eine Umsteigekante verbunden sind, erhalten wir $(|L|)$ Kanten pro Haltestelle, also $|L||V|$ insgesamt. Aufsummiert ergibt das $|E'| \leq |V||L| + (|L| - 1)|V|$. □

Falls im Linienverbund L auch Linien mit Kreisen auftauchen, kann der Change&Go Graph natürlich theoretisch beliebig groß werden. Exakt besteht er aus $|V| + \sum_{l \in L} |l|$ Knoten.
Praktisch bedeutet dieses Ergebnis, dass der C&G Graph deutlich größer als unser Ursprungsgraph G ist, jedoch nur quadratisch wächst bzw. linear in Abhängigkeit vom Linienpool L. Auch die Kantenzahl steigt nicht stark nämlich linear mit der Zahl der Linien in L und linear mit Anzahl der Knoten. Im numerischen Teil werden wir aber noch sehen, dass selbst dieses quadratische Wachstum sehr viel sein kann. Wird ein Dijkstra nämlich auf den Change&Go Graphen angewendet, der selbst quadratische Laufzeit hat, so wächst die Laufzeit schon in $\mathcal{O}(n^4)$, was nicht mehr wenig ist. Ist der Dijkstra wie bei uns in andere Algorithmen eingebettet, potenzieren sich diese Polynome.

Kommen wir nun endlich zur versprochenen Erweiterung unseres bestehenden H_0 Algorithmus:

5.2 Heuristik H

Input: Das PTN $G = (V, E)$ mit $n := |V|$ und $m := |E|$ sowie der OD-Matrix W.

1. $L = \varnothing$. $\forall e \in E$ füge (e) als Linie in L hinzu.
2. Initialisiere den Change&Go Graphen $G' = (V', E')$ und die Matchingprobleme $M_1, ..., M_n$ sowie die Variable $time = \infty$.
3. Solange $zfw\_alt > \sum_{p \in P} t(p, L)$ (also der Zielfunktionswert sich verbessert hat) führe aus:
   (a) $zfw\_alt := \sum_{p \in P} t(p, L)$
   (b) Berechne zu jedem Fahrgast seinen bevorzugten Reisepfad in G'.
   (c) Für alle Knoten $v_i \in V$ führe Subalgorithmus 1 an v_i aus.

Output: Linienverbund L.

Im Grunde wird hier der Algorithmus H_0 solange hintereinander ausgeführt, bis sich der Zielfunktionswert zfw_alt nicht mehr verbessert.

Lemma: 5.2.1 *Heuristik H verschlechtert den Zielfunktionswert eines Linienverbundes nicht bezüglich des seriellen Linienproblems.*

Beweis: Wie bei H_0 folgt dies daraus, dass der einzige Schritt, in dem der Linienverbund verändert wird, der Subalgorithmus 1 ist, für den diese Eigenschaft zutrifft. Deswegen verschlechtert sich der Zielfunktionswert auch nicht nach Ausführung des ganzen Algorithmus H. □

Lemma: 5.2.2 *Algorithmus H determiniert nach endlich vielen Schritten.*

Beweis: Die Schritte 3a) und 3b) (mitsammt den Unterschritten) entsprechen dem Algorithmus H_0, bei dem bekanntermaßen der Zielfunktionswert $\sum_{p \in P} t(p)$ nie größer sondern allenfalls kleiner werden kann. Da unser Graph ein PTN ist, hat er positive Kantengewichte und es gibt auf jeden Fall eine untere Schranke des Zielfunktionswertet, nämlich die Reisezeit der Fahrgäste ohne Umsteigekosten; kein Passagier kann schneller an sein Ziel gelangen als auf dem kürzesten Weg ohne umsteigen zu müssen. Diese untere Schranke existiert, weil wir endlich viele Fahrgäste haben.
Wir nehmen an, der Algorithmus würde nicht determinieren, also unendlich viele Schritte sich stets verbessern. Weil $|E| < \infty$, existiert ein $\varepsilon > 0$ nämlich

$$\varepsilon = \min \left\{ (\sum_{e \in E'} t_e) - (\sum_{e \in E''} t_e) \Big| E', E'' \subset E \cup \{e_r\} \textit{ mit } t_{e_r} = r \textit{ und } (\sum_{e \in E'} t_e) > (\sum_{e \in E''} t_e) \right\}$$

sodass der Zielfunktionswert sich niemals in einem Schritt um weniger als um ε verbessern kann. r ist dabei die Umsteigebestrafungszeit. Dieses ε existiert, weil es endlich viele Kanten gibt, und ist fix für alle Schritte, denn jeder Fahrgast, der seine Route verbessert, tauscht Teile seines Weges durch neue Teilwege. Wenn der

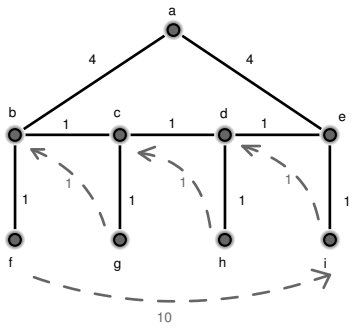
Algorithmus sich in jedem Schritt um mindestens ε verbessert, muss der Zielfunktionswert irgendwann unter die untere Schranke rutschen, was ein Widerspruch ist. ↯
Also ist unserer Annahme falsch, dass der Algorithmus nicht determiniert. □

Lemma: 5.2.3 *Sei $G = (V, E)$ ein PTN und ein kreisfreier Graph und sei die OD-Matrix W gegeben. Dann liefert H eine Optimallösung für das serielle Linienproblem ohne Budget.*

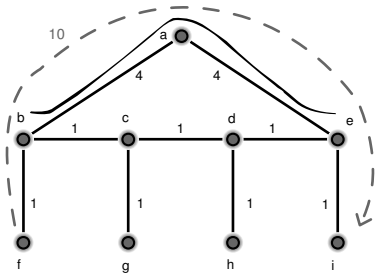
Beweis: Nachdem H einmal ausgeführt worden ist, ist der aktuelle Linienpool identisch zum Output des H_0 Algorithmus. Anschließend kann sich der Linienpool nicht mehr verschlechtern, weswegen er für kreisfreie Graphen optimal ist und bleibt. □

Für nicht kreisfreie Graphen gilt diese Aussage jedoch nicht im Allgemeinen, wie das folgende Beispiel zeigen wird:

Beispiel:



Sei nebenstehender Graph gegeben mit Knoten a bis i. Die Kantenlänge sind alle 1 bis auf die beiden zu a adjazenten Kanten, die eine Länge von 4 haben. Zudem seien OD-Paare gegeben, also Passagiere. 10 Fahrgäste wollen von Knoten f nach Knoten i. Ein Fahrgast von i nach d, einer von h nach e und ein letzter von g nach b. Als Umsteigezeit, die ein Fahrgast benötigt, um von einer Linie in eine andere zu wechseln, geben wir einfach 3 an. In der Startaufstellung bildet jede Kante eine eigene Linie, weswegen anfangs sehr viele Umsteigevorgänge nötig sein werden.

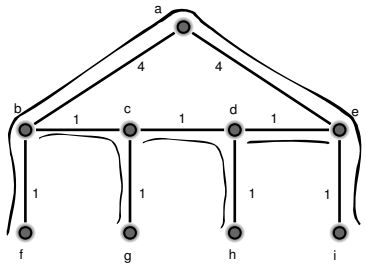


Wenn wir nun Algorithmus H anwenden, werden zuerst in der Umgebung von Knoten a die beiden Kanten mit Gewicht 4 zu einer Linie gematcht. Danach folgt für das Matchingproblem zu b, dass die Fahrgäste, die von f nach i wollen, den langen Weg über Knoten a nehmen werden, weil sie da zwei Umsteigevorgänge weniger benötigen (6 Minuten schneller), auch wenn die Wegstrecke 5 länger ist.

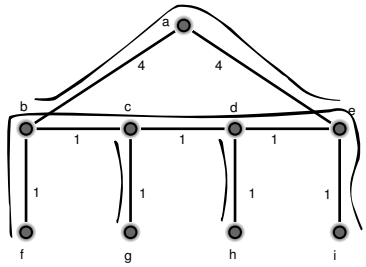
$$t((f, b, c, d, e, i)) = 17$$

$$t((f, b, a, e, i)) = 16$$

Also wird Kante {b, f} mit {a,b} zu einer Linie mit zusätzlich {a, e} gematcht, die nun drei Kanten lang ist.



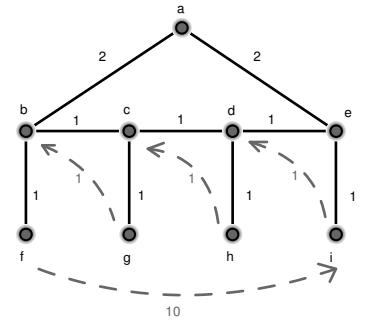
Hiernach geht der Rest einfach. Bei Knoten c werden die Kanten {b, c} und {c,g} zu einer Linie gematcht, um Knoten d werden {c, d} und {d, h} gematcht und bei Knoten f bleibt (e, f) alleine eine Linie, wohingegen (f, i) sich der langen Linie (f, b, a, e) anschließt. Das ergibt für unsere Zielfunktion einen Wert von $10 \cdot 10 + 2 + 2 + 5 = 109$.



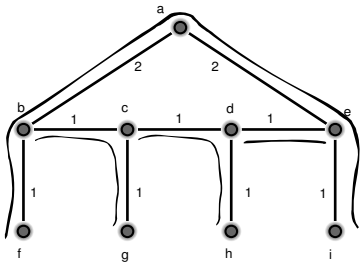
Das ist aber keineswegs optimal. Wenn es nämlich eine lange Linie (f, b, c, d, i) geben würde und zusätzlich noch (b, a, e), sowie (c, g) und (d, h), so würde der Zielfunktionswert $10 \cdot 5 + 5 + 5 + 2 = 62$ ergeben.

Ein kleines Detail könnte uns etwas fröhlicher stimmen: wenn wir nicht bei Knoten a angefangen hätten zu iterieren, sondern bei b, so wäre eben jene Optimallösung heraus gekommen, weil bereits vor dem ersten Schritt der kürzeste Weg des 10er OD-Paares über b, c, d und e gelaufen wäre. Das tat es zwar vor dem ersten Schritt auch geradeeben, aber nach dem ersten Schritt, wo über a einmal Umsteigen wegfallen würde, hat sich der kürzeste Weg geändert.
Es kann also durchaus an der Reihenfolge der Knoten liegen, ob wir eine gute oder eine weniger gute Lösung erhalten. Aber es muss nicht, wie das folgende Beispiel zeigen wird:

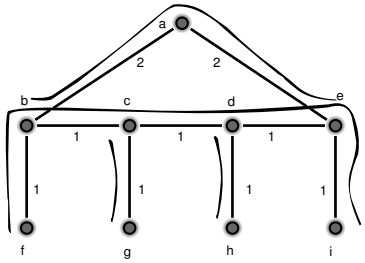
Beispiel:



Alles ist wie gerade eben, nur die Kantengewichte der oberen zwei Kanten haben sich von 4 auf 2 geändert. Der Leser mag es selbst ausprobieren: egal, in welcher Reihenfolge man den H-Algorithmus iterieren lässt, man erhält ...



... diesen Linienpool mit (f, b, a, e, i) als eine Linie und wieder (b, c, g) und (c, d, h) und (d, e) als weitere Linien. Der Zielfunktionswert ist mit diesem Pool $10 \cdot 6 + 2 + 2 + 5 = 69$.



Aber optimal wäre wieder dieser Pool mit [f, b, c, d, e, i] als großer Linie und [c, g] und [d, h] und [b, a, e] als weitere Linien, welcher den Zielfunktionswert $10 \cdot 5 + 5 + 2 = 62$ hat, also besser ist.

Es kann also passieren, dass der H-Algorithmus uns auf jeden Fall eine Optimallösung liefert wie bei kreisfreien Graphen, es kann aber auch sein, dass es von der Reihenfolge der Knoten abhängt, ob er eine gute oder weniger gute Lösung findet. Und es kann schließlich auch der Fall eintreten, dass der H-Algorithmus niemals eine optimale Lösung findet.

Lemma: 5.2.4 *Sei G ein PTN mit n Knoten und m Kanten und W die OD-Matrix. Der Algorithmus H erzeugt dann einen Linienpool genau der Größe $\frac{k}{2}$, wobei k die Anzahl aller Knoten mit ungeraden Knotengrad in G sind.*

Beweis: Hierfür muss man sich zwei Sachen klar machen. Zum einen werden alle gefundenen Matchings welche mit maximaler Kardinalität sein; es kann nämlich zwar sein, dass eine Kante des Matchings Gewicht Null hat, aber die wollen wir ohne Einschränkung der Allgemeinheit auch im Matching sehen. Demnach wird bei einem Knoten mit geradem Knotengrad nie eine Linie enden, sondern immer woanders weiter fahren. Anders bei Knoten mit ungeradem Knotengrad - da gibt es kein perfektes Matching, weswegen das gefundene Matching einen aus- bzw. eingehenden Knoten (des Subgraphen) nicht beinhaltet.
Jetzt muss man sich noch als zweites vergewissern, dass jede Linie genau zwei Enden hat. Das tut sie zweifelsohne, weil eine Linie ein Pfad ist. Demnach ist die Anzahl aller von H erzeugten Linien genau $\frac{k}{2}$.
Genau genommen gilt dies sogar nach jeder Ausführung des Subalgorithmus 1, und damit auch für H_0. □

Lemma: 5.2.5 *Sei G ein PTN mit n Knoten und m Kanten und W die OD-Matrix. Es gibt eine optimale Lösung L des seriellen Linienproblems ohne Budget, sodass gilt: $|L| = \frac{k}{2}$, wobei k die Anzahl aller Knoten in G mit ungeradem Knotengrad ist.*

Beweis: Wir nehmen an, dass wir ein optimales L haben, das diese Eigenschaft nicht besitzt. Auf G mit Linienpool L lassen wir unseren Algorithmus H operieren, also führen wir den Subalgorithmus 1 mit kürzesten Wegen und Matchings auf jeden

Knoten $v_i \in V$ aus. Heraus kommt ein neuer Linienverbund L', dessen Zielfunktionswert nicht schlechter als der von L ist (wegen Lemma 5.2.1). Also ist L' ebenfalls optimal. Zudem gilt für L', dass er genau $\frac{k}{2}$ Linien besitzt. □

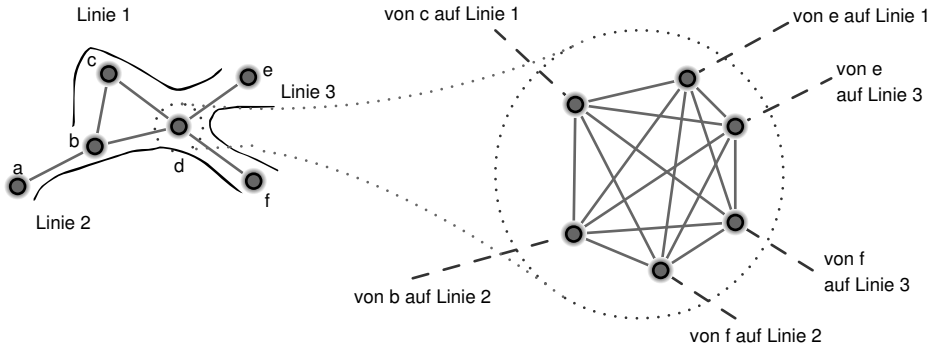
Obwohl das Beispiel oben gezeigt hat, dass bei allgemeinen also nicht kreisfreien Graphen der H-Algorithmus nicht unbedingt eine Optimallösung finden muss, so haben die zwei Lemmata gerade zumindest gezeigt, dass die Lösung des H-Algorithmus genau die Anzahl an Linien hervorbringt wie sie auch jede Optimallösung hat.
Wir sind also in manchen Fällen zwar nicht optimal, aber zumindest nicht ganz unterschiedlich zur Optimallösung. Das soll uns zumindest ein kleiner Trost sein.

Immer noch fehlt in unserem Algorithmus jede Spur von einer Kostenfunktion und der oben angesprochenen Nebenbedingung. Das hat seinen Grund weiterhin in der Problematik, dass wir derzeitig nicht sehr viel daran ändern können, dass stets genau eine Linie über eine Kante fährt. Nie können zwei Linien parallel fahren und ebensowenig könnte eine Strecke gar nicht befahren werden. Unter diesen Umständen ergibt es wenig Sinn, über Kosten nachzudenken.
Theoretisch ist das Konstrukt eines Change&Go Graphs sehr gut in der Lage, mit parallelen Linien arbeiten zu können. Also wenden wir uns nun diesem Problem zu, um in der Folge auch ein Budget mit einzuberechnen. Gewissermaßen wollen wir also statt das serielle Linienproblem ohne Budget nun das parallele Linienproblem mit Budget betrachten.

6 Split&Merge Heuristik (paralleles Linienproblem)

Bisher hat unser Algorithmus einen deutlichen Schönheitsfehler: pro Strecke existiert nur eine Linie und nie werden Linien im endgültigen Linienpool auftreten, die zumindest über Teilstrecken parallel fahren. Damit haben die bisherigen Algorithmen die schöne Eigenschaft gehabt, dass sie auf Bäume angewendet optimale Lösungen des seriellen Linienproblems ohne Budget finden. In der Praxis will man allerdings natürlich, dass mehrere Buslinien über Teilstrecken parallel verlaufen, was konkret unserem parallelen Linienproblem mit/ohne Budget entspricht. Dieses Manko werden wir in diesem Kapitel ausbessern und eine neue Heuristik vorstellen.

Wir gehen gedanklich zu unseren Matchingproblemen zurück und konstruieren uns wieder zu einer Haltestelle $i \in V$ ein Matchinggraph M_i. Die Anzahl der Knoten in M_i wurde zuvor stets als der Knotengrad von i, also die Anzahl der eingehenden Kanten zu i angenommen. Im Grunde interessiert uns aber nicht direkt die Kante, sondern eher die Linie, auf der ein Fahrgast sich bewegt. Bisher ist das ein- und dasselbe gewesen, weil über jede Kante genau eine Linie fahren sollte. Wir haben diese Linien zuerst zerschnitten, das Matchingproblem und seine Lösung berechnet und daraus die Linien neu zusammen gefügt.
Wenn wir jetzt für das parallele Linienproblem einfach wieder die Anzahl der eingehenden Linien (nach dem Zerschneiden) als $|M_i|$ nehmen, müsste alles wie zuvor funktionieren. M_i bleibt wie zuvor selbstverständlich vollständig.



Man sieht gleich, dass der vollständige Matchinggraph M_i leicht unübersichtlich werden kann. Ein Matchingproblem zu einem zentralen Busbahnhof mit 8 Linien, die zu 16 Knoten mit 240 Kanten werden, möchte kaum einer von Hand ausrechnen.

Definition: 6.0.6 *Sei ein PTN $G = (V, E)$ gegeben. Wir bezeichnen einen Graphen $M_i = (X_i, Y_i)$ als einen Linien-Matchinggraphen zu einem Knoten i, wenn gilt:*

- $$|X_i| = \sum_{l \in L} \sum_{\substack{j \leq |l| \\ i \in l(j)}} 1,$$
- *M_i ist ein vollständiger Graph,*
- *M_i hat nichtnegative natürliche Kantengewichte.*

Wir bezeichnen die Knoten von M_i mit $(l(j), h)$, wobei h eine natürliche positive Zahl ist, die nur sicher stellt, dass man die Knoten auch auseinander halten kann, wenn mehrere Linien eine zu i inzidente Kante befährt. Gibt es also drei Linien, die die Kante $\{a, b\}$ befahren, so tauchen in M_a die Knoten $(\{a, b\}, 1)$, $(\{a, b\}, 2)$ und $(\{a, b\}, 3)$ auf.

Definition: 6.0.7 *Seien ein PTN $G = (V, E)$, ein Linienverbund L und die zu G passenden Linien-Matchingprobleme $M_1 = (X_1, Y_1), ..., M_n = (X_n, Y_n)$ sowie eine Indexmenge I für L gegeben. Wir definieren die Abbildungen*

$$S_1 : \mathcal{L}_G \rightarrow \begin{pmatrix} \mathcal{M}(Y_1) \\ \vdots \\ \mathcal{M}(Y_n) \end{pmatrix}$$

$$L \mapsto \begin{pmatrix} \left\{ \{(e_j, h(l,a)), (e_k, h(l,a+1))\} \in M_1 \text{ wenn } \exists l \in L \text{ mit } l(a) = e_j \text{ und } l(a+1) = e_k \right\} \\ \vdots \\ \left\{ \{(e_j, h(l,a)), (e_k, h(l,a+1))\} \in M_n \text{ wenn } \exists l \in L \text{ mit } l(a) = e_j \text{ und } l(a+1) = e_k \right\} \end{pmatrix}$$

dabei definieren wir $h(l, a)$ einfach als Zählvariable, wobei wir die Indexierung I von L benötigen:

$$h(l_i, a) := \sum_{\substack{b \leq a \\ l_i(b) = l_i(a)}} 1 + \sum_{j<i} \sum_{\substack{b \leq |l_j| \\ l_i(b) = l_j(a)}} 1$$

und

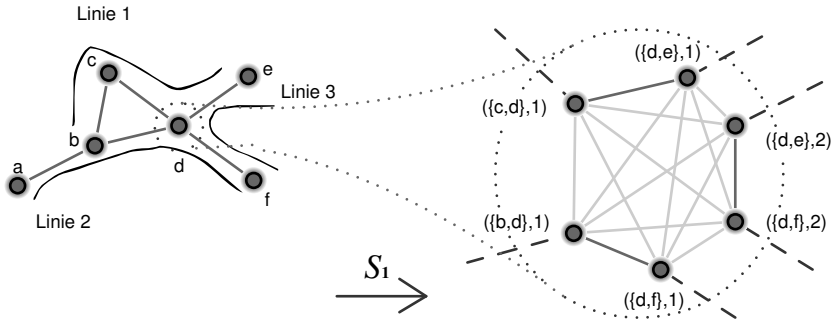
$$S_2 : \begin{pmatrix} \mathcal{M}(Y_1) \\ \vdots \\ \mathcal{M}(Y_n) \end{pmatrix} \rightarrow \mathcal{L}_G$$

$$\begin{pmatrix} m_1 \\ \vdots \\ m_n \end{pmatrix} \mapsto \left\{ (e_{i_1}, ..., e_{i_x}) \;\middle|\; \begin{array}{c} \forall 1 \leq j < x \;\; \exists e' \in \bigcup_{k=1}^{n} m_k \;\; \text{mit} \;\; e' = \{(e_{i_j}, h), (e_{i_{j+1}}, h')\} \\ \text{und} \\ \left(x > 1 \text{ oder } \nexists e' \in \bigcup_{k=1}^{n} m_k \;\; \text{mit} \;\; e' = \{(e_{i_j}, h), (e_{i_{j+1}}, h')\}\right) \end{array} \right\}$$

wobei $\mathcal{M}(Y_i)$ Menge aller Matchings der Kantenmenge Y_i ist und $l(a)$ die a-te Kante der Linie l.

Diese Definition ist fast die gleiche wie bei den Kanten-Matchinggraphen des seriellen Linienproblems, nur dass hier die Knoten des Linien-Matchinggraphen etwas anders heißen, um die Eindeutigkeit der Bezeichnung zu bewahren. Ansonsten hat die neue Bezeichnung eigentlich keine große Bedeutung. Die speziell definierte Funktion $h(l, a)$ soll sicher stellen, dass wir zwei Knoten x und y von Linien-Matchingproblemen zu zwei benachbarten Knoten M_i und M_j eine gleiche Konstante h haben, wenn x und y von der gleichen Linie stammen und damit im Matching von $S_1(L)$ auftauchen.

Die Abbildung S_1 sieht dann mit dem oberen Beispiel etwa so aus:



Man muss sich bei diesem Schaubild klar werden, was zuerst da gewesen ist - Henne oder Ei bzw. Matchinggraph oder Matching: Der Knoten $(\{d, e\}, 2)$ hat nicht etwa den zweiten Eintrag 2, weil die Linie 3 die zweite Linie ist, die die Kante $\{d, e\}$ befährt. Sondern es gibt zuerst einmal Knoten $(\{d, e\}, 1)$ und $(\{d, e\}, 2)$, weil es genau zwei Linien gibt, die diese Kante befahren. Welcher Knoten von M_d welcher Linie zugeordnet wird, ist die Aufgabe von S_1 bzw. von der in S_1 definierten Funktion h bzw. des Indexes I von L. Wäre S_1 bzw. h bzw. I anders definiert, so würde auch die Zuordnung anders sein. Dieser Umstand führt dazu, dass unsere Zuordnung zwischen Linienverbund und Matching nicht ganz eineindeutig ist.

Satz: 6.0.8 *Seien ein PTN $G = (V, E)$ und ein Linienverbund L als Lösung zum parallelen Linienproblem gegeben. S_1 ist injektiv und es gilt $S_2 \circ S_1 = id$.*

Beweis: Zuerst zeigen wir: S_1 ist wohldefiniert und injektiv.

- Für die Wohldefiniertheit nehmen wir an, dass es einen Linienverbund L gebe, sodass eine der Komponenten von $S_1(L)$ kein Matching ist. Dann würde es in dieser Komponente m_k zwei Kanten e_i und e_j geben, die einen gemeinsamen Knoten haben. Daraus folgt, dass es laut Vorschrift eine Linie in L gibt, für die gilt $l_1(a) = l_1(a)$. Das gilt natürlich, aber nicht für verschiedenes a. Durch die zusätzliche Bezeichnung der Knoten in $|M_i|$ ist die Wohldefiniertheit also besonders gewährleistet.

- Injektivität: Gegeben seien zwei Linienverbünde L_1 und L_2 mit $S_1(L_1) = S_1(L_2) = (m_1, ..., m_n)$. Für jede Kante $\{(e_i, l, a), (e_j, l, a + 1)\}$ in $\bigcup_{k=1}^{n} m_k$ gibt es sowohl in L_1 als auch in L_2 eine Linie, die e_i und direkt danach e_j überfährt. Weil das für alle Kanten in m_i gilt, können sich die beiden Linienverbünde zumindest an i nicht unterscheiden. Wenn man dies für alle Knoten $i \in V$ einmal durchexerziert, folgt, dass L_1 und L_2 insgesamt gleich sein müssen (bis auf Inversion der Linien, also Vertauschung der Anfangs- und Endpunkte).

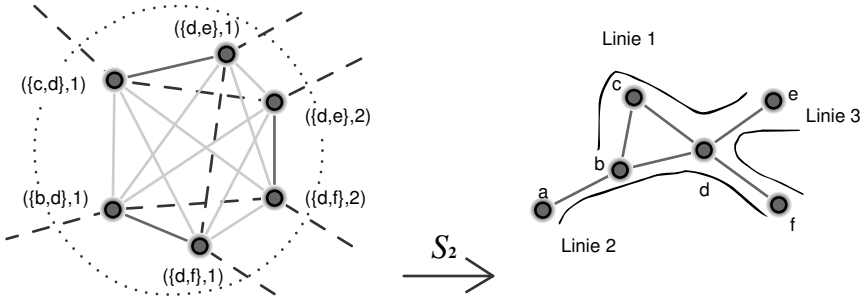
Jetzt behandeln wir noch S_2:

- Wohldefiniertheit: Zuerst muss gezeigt werden, dass S_2 auch wirklich einen Linienverbund erzeugt aus den Matchings. $S_2((m_1, ..., m_n))$ ist eine Menge von Vektoren mit Kanten als Komponenten. Jeweils zwei aufeinanderfolgende Kanten inzidieren, weil es eine Kante in einem Matching gibt, sodass beide Kanten Element der Matchingkante sind. Demnach sind die Vektoren von Kanten auch gültige Pfade und damit $S_2((m_1, ..., m_n))$ ein Verbund von Linien. Damit ist also auch die Wohldefiniertheit gezeigt.

Jetzt können wir S_2 als Umkehrabbildung zu S_1 auffassen: Es gilt, dass $S_2 \circ S_1 = id$. Und das liegt gerade daran, dass aus einer Kantenreihenfolge (x, y) von einem $l \in L$

eine Matchingkante $\{(x, h(l, a)), (y, h(l, a + 1))\}$ in einem Matching wird.
Diese Matchingkante wird durch S_2 wieder genau den Kanten zugeordnet, also x und y. Also gibt es im Bild von S_2 wieder eine Linie, die nacheinander genau x und y anfährt. Durch die allgemeine Definition von $h(l, a)$ ist auch gewährleistet, dass an den adjazenten Knoten zu x und y ebenfalls die Linien genau so weiter fahren wie zuvor. Also bleibt die Kantenreihenfolge jeder Linie beibehalten, was wieder den gleichen Linienverbund ergibt. Damit gilt also auch $S_2 \circ S_1 = id$ und natürlich auch $S_1 \circ S_2 = id$. □

Was uns fehlt im Gegensatz zum seriellen Linienproblem ist die Injektivität von S_2. Die ist aber gerade nicht gültig, weil durch Veränderungen des Parameters h in den Knoten des Linienmatchingproblems trotzdem ein- und derselbe Linienverbund entstehen kann. Oben haben wir gesagt, dass die Definition von h auch anders hätte aussehen können. Jetzt stellen wir fest, dass das Ergebnis von S_2 dann trotzdem gleich geblieben wäre. Das folgende Bild zeigt das noch einmal deutlich: sowohl das normale Matching von $S_1(L)$ als auch das gestrichelte Matching führen beide zur gleichen Lösung, die unser ursprünglicher Linienverbund L ist. Oder noch anders ausgedrückt: es gilt zwar $S_2 \circ S_1 = id$, aber es kann sein, dass $S_1 \circ S_2 \neq id$ ist (das kommt dann auf h an).



In Kapitel 5 hatten wir nach der Aussage der Bijektivität von S_1 noch eine zusätzliche Aussage über die Menge aller Linienverbünde treffen können, die das serielle Linienproblem zu G löst. Weil bei unserem parallelen Linienproblem S_1 nur injektiv und nicht surjektiv ist (was ja äquivalent zur Injektivität von S_2 ist), geht das nun nicht so schön. Aber statt dessen können wir das mit Lemma 5.1.2 bewerkstelligen.

Lemma: 6.0.9 *Sei ein PTN G gegeben. Die Anzahl der möglichen Lösungen des parallen Linienproblems ist gleich der Anzahl aller Change&Go Graphen zu G.*

Beweis: Dies folgt aus der in Lemma 5.1.2 definierten Abbildung T_G, die bijektiv ist. Weil sie bijektiv ist, bildet sie zwischen gleichmächtigen Mengen ab. □

Diese Ausage ist jedoch keinesfalls für eine konkrete Abschätzung nützlich, weil beides - die Menge aller Change&Go Graph zu G und die Menge aller möglichen Linienverbünde, die das parallele Linienproblem zu G lösen können - unendlich sind. Denn schon in einem PTN mit nur einer Kante kann eine Linie beliebig häufig diese Kante hintereinander abfahren - jede zusätzliche Wiederholung ist im Grunde schon eine neue Linie und damit ein neuer Linienverbund. Diese Aussage ist daher keinesfalls so schön wie die im vorherigen Kapitel.

Nun haben wir S_1 als eine injektive Abbildung, die wir umkehren können. Diese Aussage reicht uns, um eine neue Version des Subalgorithmus anzuführen:

Lemma: 6.0.10 *Seien ein PTN $G = (V, E)$ und ein Linienverbund L sowie der C&G Graph G' gegeben. Für jeden Fahrgast in P ermitteln wir einen (nicht alle) kürzesten Weg $p = (p_1, ..., p_x)$ in G'. Betrachten wir ein zu einer Haltestelle i zugehöriges Linien-Matchingproblem M_i und seien die Kantengewichte dieses Linien-Matchingproblems gerade*

$$t_e = |\{p|\ halt(p_x) = i \text{ und } (\{halt(p_{x-1}), halt(p_{x+1})\}, h(l, a)) \in M_i\}|$$

für alle $e \in M_i$. Dann gibt es für die Fahrgäste im Linienverbund L genau so viele Umsteigevorgänge, wie das aufsummierte Gewicht aller Kanten beträgt, die nicht Element einer Komponente von $S_1(L) = (m_1, ..., m_n)$ sind.

Beweis: Jeder Fahrgast, der als Zahl auf einer ungematchten Kante durch seinen kürzesten Weg hinzugefügt wurde, muss an der Haltestelle einmal umsteigen, weil es keine Linie gibt, die den selben Weg nimmt wie der Fahrgast. Auf der anderen Seite, muss kein Fahrgast, der als Zahl auf einer gematchten Kante hinzugefügt wurde, an der Haltestelle umsteigen, weil in L eine Linie existiert, die den Weg des Fahrgastes durchfährt. Also müssen an jeder Haltestelle genau so viele Personen umsteigen, wie das Gewicht aller Kanten beträgt, die nicht im Matching sind. □

6.1 Subalgorithmus 2

Input: PTN $G = (V, E)$ mit OD-Matrix W und einem Linienverbund L auf G, einem spezifischen Knoten $v \in V$ und zu jedem Fahrgast p einen für ihn günstigsten Pfad, in dem über das PTN reisen möchte.

1. Erzeuge Linien-Matchingprobleme $M_1, ..., M_n$ und Matchings $m_1, ..., m_n$ durch $S_1(L)$;
2. trage anhand der kürzesten Wege der Fahrgäste die Kantengewichte in M_v auf;
3. berechne m_v für M_v neu als gewichtsmaximales Matching;
4. $L' = S_2((m_1, ..., m_v, ..., m_n))$.

Output: Linienverbund L'.

Die verwendeten S_1 und S_2 sind selbstverständlich die soeben entwickelten und nicht die, die im Subalgorithmus 1 zum Zuge kamen.

Satz: 6.1.1 *Sei $G = (V, E)$ ein PTN mit der OD-Matrix W und zudem ein Knoten $v \in V$ gegeben. Der Subalgorithmus 2 verschlechtert den Zielfunktionswert bezüglich des parallelen Linienproblems ohne Budget nicht.*

Beweis: Bevor wir den Subalgorithmus 2 einmal anwenden, teilen wir unsere Fahrgäste auf in Fahrgäste, deren ermittelter kürzester Weg über Knoten v verläuft. Wir behalten aber auch im Kopf, dass diese kürzesten Wege nicht eindeutig sein müssen. Von jedem Fahrgast zählt nur ein kürzester Weg und ob ein eventuell anderer gleichlanger Weg über Knoten v laufen würde, können wir jetzt nicht wissen.

$$\sum_{p \in P} t(p, L) = \sum_{p \in P_v} t(p, L) + \sum_{p \in P \setminus P_v} t(p, L)$$

P_v ist nun die Menge aller Fahrgäste, deren ermittelter kürzester Weg über v läuft. Jetzt wenden wir den Subalgorithmus an. Naturgemäß sind alle $p \in P_v$ gerade die Kantengewichte des Linien-Matchingproblems M_v.
Nehmen wir nun an, dass der Subalgorithmus den Zielfunktionswert verschlechtert hat. Die Verschlechterung kann nicht in der Gruppe der Fahrgäste $P \setminus P_v$ aufgetaucht sein, weil deren kürzeste Wege unverändert blieben (es kann sein, dass ihr neuer kürzester Weg über v verläuft, aber das liegt dann keinesfalls daran, dass sich ihr Weg verschlechtert hat, weil der ursprüngliche Weg noch gleich ist). Also müssen sich die Wege von P_v verschlechtert haben. Daraus resultiert, dass nach der Ausführung des Subalgorithmus mehr Fahrgäste umsteigen müssen, denn die eigentliche Reisezeit hat sich ebenfalls nicht geändert (die Kantengewichte blieben nämlich gleich). Das ist aber ein Widerspruch zu Lemma 6.0.10, denn das gelöste gewichtsmaximale Matchingproblem garantiert gerade, dass in der Summe möglichst wenige Fahrgäste umsteigen müssen. Und weil kein Fahrgast jemals zweimal an einer Haltestelle umsteigen muss, sinkt dadurch die gesamte Bestrafungstzeit der Fahrgäste $p \in P_v$. Also kann sich die Zielfunktion nur verbessern oder gleich bleiben. □

6.2 Split & Merge Heuristik mit fester Kantenkardinalität

Wir haben als Input im Gegensatz zum H-Algorithmus noch einen Linienpool L. Dieser Linienpool kann auch so aussehen, dass Linien streckenweise parallel verlaufen. Der folgende Algorithmus bietet eine einfache Optimierung dieses Pools.

Split&Merge mit vorgegebenem Linienverbund

Input: Ein PTN G mit der OD-Matrix W und einem Linienverbund L.

1. Initialisiere den Change&Go Graph $G' = (V', E')$ und die n Matchingprobleme $M_1, ..., M_n$ mit Beachtung der doppelten Linien pro Kante. Setze $time := \infty$.
2. Solange $zfw\_alt > \sum_{p \in P} t(p, L)$ tu
   (a) $zfw\_alt = \sum_{p \in P} t(p, L)$.
   (b) Für alle $v_i \in V$ berechne L neu durch Ausführen des Subalgorithmus 2 an Haltestelle v_i.

Output: Linienverbund L.

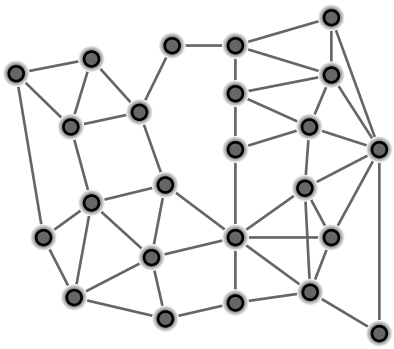
Dieser Algorithmus ist geeignet, um einen bisherigen Linienverbund zu verbessern. In gewisser Weise sind diese Verbesserungen einfacher Natur - es wird kein fundamental anderer Pool präsentiert, sondern nur ein Pool, in dem einige Linien eventuell etwas anders verlaufen. Die Veränderungen sind alle „lokal“ und nicht „global“.

Etwas mehr erreicht man, wenn man den anfänglichen Verbund in seine einzelnen Streckenabschnitte zerteilt. Das bedeutet, aus einer Linie, die über vier Kanten verläuft werden vier Linien, die jeweils über eine Kante verlaufen. Der Vorteil liegt hierbei in einer gewissen Unvoreingenommenheit der Fahrgäste. In dem Fall, dass die Linien nicht zerteilt werden, bewegen sich die Fahrgäste schon am Anfang lieber

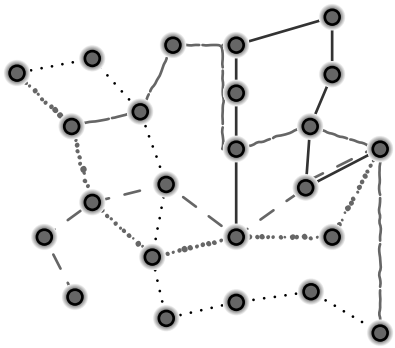
auf einer langen Linie, die einen Umweg fährt als drei Linien, die zusammen direkt zum Ziel fahren, weil die Fahrgäste von Anfang an Umsteigevorgänge einsparen. Wenn man die Linien zerschneidet, werden die Fahrgäste im ersten Schritt einen kürzesten Weg nehmen, der eingermaßen direkt zu ihrem Ziel führt, weil sie so oder so viel umsteigen müssen.
Aber wir wollen das zusätzliche Zerschneiden weniger als neuen Algorithmus anführen, sondern eher als eine Bemerkung, dass man den Linienpool, der als Input dient, möglichst unwillkürlich gestalten sollte. Denn durch eine durchfahrende Linie hier und eine unterbrochene Linie dort wird den Fahrgästen schon eine willkürliche Ordnung vorgegeben. Es kann sein, dass diese Ordnung ein optimaler Pool gewesen ist; dann würde selbstverständlich diese Ordnung durch das vorherige Zerschneiden zunichte gemacht und nur ungewiss wieder ein optimaler Pool von dem Algorithmus gefunden werden. Das Zerschneiden kann also auch negativen Effekt haben. Im Allgemeinen würde man aber eher eine Verbesserung erwarten.

6.3 Minimale Aufspannende Bäume

Im nächsten Abschnitt wollen wir einen Algorithmus entwickeln, der auf einem PTN einen Linienverbund mit mehreren parallelen Linien erzeugen kann. Dazu soll er erst einmal in einem ersten Durchlauf den Transport aller Passagiere gewährleisten und nacheinander, bis ein bestimmtes Budget erschöpft ist, immer mehr Linien in den Verbund hinzufügen. Dazu wollen wir unseren Algorithmus mit einem möglichst kleinen Startverbund beginnen, der sehr günstig ist also wenig Budget verschlingt. Zuvor gingen wir stets davon aus, dass auf jeder Kante $e \in E$ des Graphen auch genau ein Bus fahren soll. Betrachtet man das Problem aber allgemein und will ein möglichst unvoreingenommenen Input (also einen mit möglichst wenigen Einschränkungen also möglichst vielen Möglichkeiten) haben, so muss man erlauben, dass in E mehr Kanten sind als wirklich tatsächlich befahren werden sollen. Und auf der anderen Seite können in der Lösung einige Kanten natürlich auch doppelt oder mehrfach von Linien befahren werden.



PTN G mit allen Kanten



möglicher Liniengraph zu G

Beispiel: So sind die Busse in den Stadtrandbezirken leicht in der Lage, entlang des Stadtrandes in einen anderen Stadtteil zu wechseln und so kreisförmig um die Stadt zu fahren. Aber in der Praxis passiert das nicht. Ob diese Vorgehensweise günstig ist oder nicht, sei dahin gestellt. Aber Fakt ist, dass wir den Graphen um eine Möglichkeit bereichern, wenn wir die mögliche Verbindung einer Stadtrandverbindung in den Graphen einbinden - auch wenn er in der Lösung nicht vorkommen muss. Schließlich ändert sich ja nichts an unserem Budget.
Für unseren Algorithmus bedeutet das, dass er nicht alle Kanten in seinen Linienpool aufnehmen soll, sondern nur eine geringe Auswahl, die aber trotzdem ausreicht,

alle Passagiere (aus unserer OD-Matrix) an ihren Zielort zu bringen. Für diese kleine Auswahl verwenden wir einen Minimalen Aufspannenden Baum.

Definition: 6.3.1 *Sei $G = (V, E)$ ein gewichteter Graph und $T \subset E$ ein Baum. T ist aufspannend, wenn es für alle $v \in V$ ein $e \in T$ gibt, sodass $v \in E$. T heißt Minimaler Aufspannender Baum (oder MST aus dem Englischen), wenn es keinen aufspannenden Baum T' gibt mit geringerer Summe der Kantengewichte.*

Ein solcher MST (minimal spanning tree) lässt sich in linearer Laufzeit ermitteln - zum Beispiel mit dem Algorithmus von Kruskal. Wir wollen diesen MST als grundlegenden Linienverbund für unsere Algorithmen verwenden, indem jede Kante des MST eine einzelne Linie im Verbund darstellt.
Leider kann man nicht davon ausgehen, dass der MST eines Graphen auch wirklich im optimalen Linienverbund L zum parallelen Linienproblem enthalten ist.

6.4 Split & Merge Algorithmus mit dynamischer Kantenkardinalität

Im Folgenden wollen wir ausgehend von einem kleinen Linienverbund wie dem MST Split & Merge ausführen. Danach erweitern wir den Pool um eine Linie, führen Split&Merge um ein weiteres Mal aus, bis er determiniert. Danach erweitern wir den Pool wieder und so weiter, bis das Budget ausgereizt ist. Also wird hauptsächlich einmal der H-Algorithmus angewandt, dann einmal der Pool erweitert, dann der H-Algorithmus eingesetzt und so weiter. Ein recht naheliegender Ansatz.

6.5 Split & Merge Algorithmus mit Budgetgrenze

Input: Ein PTN G mit der OD-Matrix W und einem Budget $b \in \mathbb{R}$, sowie einer Kostenfunktion $c : \mathcal{L}_G \to \mathbb{R}$.

1. $L = \varnothing$; Berechne minimalen spannender Baum T; $\forall e \in T$ füge Weg (e) zu L hinzu.
2. Initialisiere den Change&Go Graph $G' = (V', E')$ und die n Matchingprobleme $M_1, ..., M_n$ und $time = \infty$.
3. Solange $zwf\_alt \neq \sum_{p \in P} t(p, L)$ tu
   (a) $zwf\_alt = \sum_{p \in P} t(p, L)$.
   (b) Für alle $v_i \in V$ berechne L neu durch Ausführen des Subalgorithmus 2 an Haltestelle v_i.
   (c) Wenn $c(L) < b$ erweitere L um maximal $b - c(L)$ zusätzliche Kosten.
4. Output: Linienpool L.

Lemma: 6.5.1 *Split & Merge verschlechtert sich in keinem Schritt bezüglich des parallelen Linienproblems mit Budget.*

Beweis: Der Beweis geht analog zu den letzten Lemmata, die ähnlich lauteten. Punkt 3b) ist unser altbekannter Subalgorithmus, der den Linienverbund stets nicht verschlechtert. Kann also nur das Erweitern des Linienverbunds (Punkt 3c) den Linienverbund verschlechtern. Da aber alle Wege in G' auch nach dem Erweitern weiterhin möglich sind und nur zusätzliche Wege denkbar sind, wird sich die Zielfunktion der Fahrgäste auch dadurch nicht verschlechtern. □

6.6 Erweitern des Verbunds

Ein wichtiger Punkt im Split&Merge Algorithmus ist das Erweitern des Linienverbunds. Dies ist für sich genommen ein überraschend schweres mathematisches Problem.
Wir suchen eine Funktion, die als Input einen Linienverbund L, eine OD-Matrix W und ein Restbudget $b' = b - c(L)$ bekommt und uns eine Linie wieder zurück gibt, die gut zu unserem Linienverbund L passt und das Budget b nicht übersteigt. Diese Aufgabe ist vielleicht noch leicht zu lösen. Und wenn man die optimale Linie haben möchte, die das macht, wird man ebenfalls nicht gerade an die Grenzen der modernen Rechnerleistung gelangen.
Das Problem liegt aber auch nicht in dem Moment, in dem man den Verbund erweitert, sondern im Finden eines optimalen Verbundes für das parallele Linienproblem mit Budget.
Wenn man nämlich versucht, in ein paar Schritten des Split&Merge Algorithmus auf einen optimalen Linienverbund zu kommen, steht man vor einer Art Rucksackproblem. Allerdings vor einem, bei dem man von den Gegenständen (den Linien), die man in den Rucksack legen will, nicht weiß, welchen Wert sie haben. Der Wert ergibt sich erst durch die kürzesten Wege der Fahrgäste und erst durch das Zusammenspiel der Linien miteinander. Eine Berechnung der Sinnhaftigkeit der einen oder anderen Linie lässt sich da zu keinem Schritt machen. Auf jeden Fall muss die in einem Schritt beste hinzugenommene Linie nicht auch für das große Problem optimal sein.
Beim normalen Rucksackproblem kann es sich lohnen, manche Gegenstände doppelt im Rucksack zu haben. Bei unserem Linienproblem bringt es gar nichts, eine schon vorhandene Linie um ein weiteres Mal in unseren Verbund zu tun.

Unterschiede: Rucksackproblem & Erweitern des Linienverbundes

Rucksackproblem	**Erweitern des Linienverbundes**
Menge von Gegenständen, die einen festen Wert haben.	Menge von Linien, deren Wert sich erst zusammen ergibt.
Gegenstände haben feste Kosten. Gesamtkosten aller Gegenstände darf festes Budget nicht überschreiten.	Linien haben feste Kosten. Gesamtkosten aller Linien darf festes Budget nicht überschreiten.
Nur ganzzahlige Gegenstände sind erlaubt.	Ebenfalls sind nur ganzzahlige Linien erlaubt.
Unbegrenzt viele Gegenstände jeder Sorte erlaubt.	Es lohnt sich gar nicht, eine schon vorhandene Linie ein weiteres Mal in den Verbund aufzunehmen, weil dadurch die Fahrzeiten der Fahrgäste nicht besser werden. Also ist jede Linie im Grunde nur einmal verwendbar.
Menge aller verschiedenen Gegenstände ist meistens übersichtlich.	Die Menge aller verschiedenen Linien ist gigantisch groß - schon bei kleinen PTNs.

Tatsächlich wollen wir das Problem des Erweitern des Linienverbundes nur heuristisch lösen. Alles Andere würde den Rahmen des Rechenbaren bei Weitem sprengen.
Es sollen hier lediglich ein paar Ideen beschrieben werden, wie eine Erweiterung sinnvoll aussehen kann. Wir verabschieden uns hier ganz von dem Vorsatz, eine optimale Lösung für das parallele Linienproblem mit Budget zu bekommen. Wir wollen ab jetzt nur noch eine Lösung erarbeiten, die uns sinnvoll erscheinen, sodass der Zielfunktionswert über kurz oder lang sehr gut wird durch das Erweitern:

- Man kann ermitteln, welches OD-Paar am meisten umsteigen muss und aus diesem OD-Paar mit einem kürzesten Weg im eigentlichen PTN eine Linie erzeugen, die dann die Erweiterung des Verbunds ist, wenn sie denn vom Budget her noch passt.

- Man kann schauen, welches OD-Paar am meisten Umsteigevorgänge multipliziert mit seiner Fahrgastzahl hat und daraus eine Linie bauen wie oben. Das ist natürlich in jedem Fall sinnvoller. Dennoch wird dabei eigentlich an der Idee des Split&Merge Algorithmus vorbei gebaut, dass die Linien zusammen einen guten Verbund ergeben sollen. Mit diesem Ansatz werden in jedem Schritt gewissermaßen die besten Einzelkämpferlinien zum Pool hinzugefügt.

- Man kann einfach greedy versuchen herauszufinden, welche Linie die beste ist, indem man jede direkte Linie (also den kürzesten Weg zwischen zwei Knoten) einmal als Erweiterung ausprobiert und die beste nimmt. Dieses Verfahren hat Vor- aber auch Nachteile gegenüber dem gerade erwähnten Verfahren. Zum Vorteil gereicht, dass die Verbesserungskurve gerade in den ersten Schritten besser ist. Aber es kann passieren, dass in jedem Verbesserungsschritt Linien ausgewählt wurden, die vielleichts stets nur Kompromisslösungen sind für mehrere OD-Paare. Wenn das in jedem Schritt geschieht, hätte es unter Umständen am Ende besser sein können, in jedem Schritt einfach ein OD-Paar nach dem Anderen auf dem direkten Weg zum Ziel zu bringen, womit die Umsteigezahl dieses OD-Paar prompt Null ergeben hätte. Aber der Vorteil ist, dass diese Kompromisslösungen der Idee des Split&Merge Algorithmus dienlicher sind als direkte Linien für einzelne OD-Paare.

- Man könnte auch zuerst ermitteln, für wie viele Linien das Budget eigentlich ausreicht. Dann versucht man, mit der ersten dieser Linien möglichst viele OD-Paare auf einmal direkt zu bedienen, so dass sie ohne Umsteigen zum Ziel kommen. Dabei kann es natürlich vorkommen, dass die Linie keineswegs einen direkten Weg abfährt. Aber dadurch werden Kosten gespart und es können in der Summe mehr OD-Paare ohne Umsteigen ans Ziel kommen. Man versucht damit lokal die Direktfahrer zu maximieren.

- Man kann auch den Change&Go Graphen um ein weiteres Basisnetz erweitern, das nur virtuell existiert. Dieses Basisnetz umfasst genau jede Kante $e \in E$ einmal als vollwertige Linie, jedoch mit Umsteigezeit r' mit $r' = \frac{r}{2}$. Es ist für die Fahrgäste also billiger, in das Basisnetz umzusteigen als in eine normale Linie. Allerdings können sie sich nicht dauerhaft günstig durch das Basisnetz bewegen, weil auf längeren Strecken sehr viele Umsteigevorgänge nötig sein werden - jede Linie ist ja nur eine Kante lang.
  Dann werden einmal alle kürzesten Wege der OD-Paare ermitteln und gezählt, welche Linie des Basisnetzes am häufigsten benutzt wurde. Diese Linie wird in den Linienverbund hinzugefügt, das Basisnetz aus dem Change&Go Graphen gelöscht und der Split&Merge Algorithmus weiter ausgeführt bis zum nächsten Erweitern des Pools.
  Dies hat einerseits den Vorteil, dass der Pool in kleinen Schritten erweitert wird, das Budget also nur langsam aufgebraucht wird. Zum anderen ist es sinnvoll, dass dadurch Linien langsam wachsen und nicht auf einmal hinzugefügt werden. Dadurch kann der neue Verbund dynamisch wachsen und geht leichter Kompromisse ein als in den anderen Verfahren.

- Der Verbund wird von genetischen Algorithmen erweitert. Dabei könnte ein Verbund ein Individuum sein. Mehrere Verbunde bilden dann eine Population. Das Crossover kann realisiert werden durch ein Austauschen von Linien zwischen zwei Linienpools. Unklar ist hierbei allerdings, inwieweit dies noch in

einem Split&Merge Algorithmus einzubetten ist. Der bisher verwendete Verbund, der mit Split&Merge teuer optimiert wurde, kann durch das Crossover rüde zunichte gemacht werden. Also scheint diese Methode eine gute Idee, aber an dieser Stelle weniger hilfreich.

Diese Liste ist natürlich auf keinen Fall vollständig. Aber sie illustriert, vor welchem Problem wir stehen und auf was man alles achten kann/muss.

Ein großes Problem hierbei ist, dass wir nur raten können, welche Linien sich beim schrittweise Erweitern für die Fahrgäste als besonders nützlich heraus stellen. Im nächsten Kapitel wird daher ein neuer Ansatz vorgestellt, der nicht versucht, einen Linienpool von unten aus aufzubauen, sondern statt dessen einen großen Linienverbund nimmt und ihn schrittweise verkleinert. Der Vorteil ist, dass die für die Fahrgäste im Zusammenspiel besonders sinnvollen Linien erhalten bleiben.

7 Cutting-Down Heuristik

Nun wollen wir uns eine weitere Heuristik anschauen, die zwar ihre Macken hat, aber dafür deutlich einfacher ist als der Split&Merge Heuristik. Wir werden für sie wieder den Change&Go Graphen benötigen und küerzeste Wege der Passagiere, nicht jedoch Matchinggraphen und deren Lösungen.

7.1 Cutting-Down Heuristik

Input: PTN $G = (V, E)$ mit OD-Matrix W und einem Budget $b \in \mathbb{R}$.

1. Erzeuge kompletten Linienpool L mit allen kreisfreien Linien und einfachen Kreisen, sowie eine Indexmenge $I \subset \mathbb{N}$ von L.
2. Erzeuge Change&Go Graphen G' aus L.
3. Solange $c(L) > b$ führe aus:
   (a) Für alle $l_i \in L$ bestimme $k_i := \sum_{p \in P} t(p, L \setminus \{l_i\})$.
   (b) Bestimme $\min_I\{k_i\}$ und streiche entsprechend $L := L \setminus \{l_i\}$, sowie $I := I \setminus \{i\}$.

Output: Linienverbund L.

Lemma: 7.1.1 *Die Cutting-Down Heuristik findet im Allgemeinen keine Optimallösung für das parallele Linienproblem mit Budget - nicht einmal bei geeigneter Indexmenge I für L.*

Beweis: Ohne Einschränkung der Allgemeinheit nehmen wir an, dass bei Schritt 2b) des Cutting-Down Algorithmus bei gleichen $k_i = k_j$ Einträgen für $i \neq j$ von I den größeren Index auswählt und entsprechend eine Streichung vornimmt. Also bei gleichen Zielfunktionswerten wird vorzugsweise eine im Index weiter hinten stehende Linie gestrichen. Einen Unterschied für den Algrorithmus macht das keinen. Doch wenn wir den Index so wählen, dass ein optimaler Linienpool $\widehat{L}$ für das parallele Linienproblem mit Budget am Anfang des Indexes auftauchen würde, könnte man meinen, dass Cutting-Down alle Linien löscht bis auf die von $\widehat{L}$. Es wäre dann ein vermutlich NP-Vollständiges Problem, eine geeignete Indexierung I zu finden. Leider funktioniert auch das nicht.
Wir beweisen das Gegenteil durch ein Gegenbeispiel. Sei G ein *vollständiger* Graph und W eine Matrix mit Einsen auf dem gesamten oberen Dreieck und das Budget b so klein, dass nur ein kleiner optimaler Linienpool $\widehat{L}$ hinein passt, bei dem die Fahrgäste nicht ohne Umsteigen an ihr Ziel kommen. Gehen wir aber davon aus, dass im kompletten Linienpool L des anfänglichen Cutting-Down Algorithmus auch zwei Linien l_x, l_y auftauchen, die zusammen alle Kanten von G befahren, wodurch sie alle Fahrgäste in kürzester Zeit und sehr direkt an ihr Ziel kommen. $\{l_x, l_y\}$ ist somit besser im Zielfunktionswert als $\widehat{L}$, was natürlich nur erklärbar ist, wenn auch gilt $c(\{l_x, l_y\}) > b$. Gelte zudem noch $c(\{l_x\}), c(\{l_x\}) <= b$, sowie

$$\sum_{p \in P} t(p, \widehat{L}) < \sum_{p \in P} t(p, \{l_x\}), \sum_{p \in P} t(p, \{l_y\}).$$

Dann würde die Abarbeitungsliste irgendwann so aussehen:

Linie	zukünftiger ZfW ohne Linie
l_x	Optimum
l_y	Optimum
$l_1 \in \widehat{L}$	Optimum - x = untere Schranke
$l_2 \in \widehat{L}$	Optimum - x = untere Schranke
$\vdots$	$\vdots$
$l_{\lvert\widehat{L}\rvert} \in \hat{L}$	Optimum - x = untere Schranke

Die untere Schranke „Optimum - x“ kann zwar nicht erreicht werden mit dem Budget; das „weiß“ das Programm aber gewissermaßen nicht. Ganz egal wie die Indexmenge des Cutting-Down Algorithmus aussieht, es werden eher Linien aus $\widehat{L}$ gestrichen als l_x oder l_y, weil durch das Streichen von l_x bzw. l_y einige Passagiere zwangsläufig umsteigen müssen. So geschieht es, dass alle Linien von $\widehat{L}$ gestrichen werden, bis nur noch l_x und l_y (oder vergleichbare Linien) in L sind. Von denen wird noch weiter gestrichen, bis nur noch eine Linie alleine in L ist, die dann aber einen schlechteren Zielfunktionswert hat als $\widehat{L}$ Also hat Cutting-Down nicht eine Optimallösung für das parallele Linienproblem mit Budget gefunden. □

Das Problem liegt hier daran, dass die Kosten der Linien nicht in die Minimums-Funktion von 2b) eingeht. Aber auf der anderen Seite: wie sollte das vonstatten gehen?
Wenn alle Linien gleich groß wären, also gleich viel kosten würden, würde das Lemma in seiner positiven Version überdies immer noch nicht gelten. Dann würde der optimale Linienpool zwar nur aus einer Linie l_i bestehen, die aber alleine wäre vielleicht besser als l_x oder l_y alleine, die zusammen aber besser sind als l_x mit l_i bzw. l_y mit l_i, sodass auf jeden Fall l_i aus L gestrichen wird.
Wir scheitern hier wieder einmal an dem Problem, dass wir im Grunde ein Rucksackproblem vor uns haben, bei dem wir nicht wissen, wie viel die Gegenstände wert sind, die wir einpacken könnten. Der Wert der Linien ergibt sich erst im Zusammenspiel mit den anderen Linien.

Interessant ist hier vielleicht, dass man Cutting-Down auf alle diese Rucksackprobleme mit dynamischem Wert der Gegenstände (also wo sich der Wert des Rucksacks erst im Nachhinein ergibt) anwenden kann; es wird auf die Eigenschaften eines PTNs nicht gesondert eingegangen, wie es der Split&Merge Algorithmus mit den lokalen Optierungen natürlich tut. Aber auch in einem anderen Punkt hat Cutting-Down einen großen Nachteil, und das ist der Speicher, der für den kompletten Linienpool L nötig ist:

7.2 Anzahl aller möglichen Linien

Lemma: 7.2.1 *Sei $G = (V, E)$ ein ungerichteter Graph. Die Anzahl aller möglichen kreisfreien Linien in G ist höchstens $\frac{1}{2}\sum_{i=2}^{n-1} \frac{n!}{(n-i)!}$.*

Beweis: Dies ist eine sehr schlechte Abschätzung. Jeder Graph G ist Teilgraph des vollständigen Graphen $H = (V, E^+)$. Weil $G \subset H$, hat G weniger oder gleich so viele Linien wie H. In H ist allerdings die Menge aller kreisfreier Linien sehr leicht zu bestimmen. Zuerst haben wir verschiedene Start und Endpunkte A und B, wobei die Richtung egal ist, was $\frac{n(n-1)}{2}$ Möglichkeiten für direkte Wege ergibt. Summiert

man dazu noch alle Wege über einen dritten Punkt C und dazu alle Wege über zwei Umwegpunkte C und D und so weiter, so erhält man:

$$\frac{n(n-1)}{2} + \frac{n(n-1)(n-2)}{2} + ... + \frac{n!}{2} = \frac{1}{2}\sum_{i=2}^{n-1}\frac{n!}{(n-i)!}$$

Dies ist eine obere Schranke aller kreisfreien Linien in H und damit auch in G. □

Dies ist natürlich keine besonders gute Aussage, weil sie uns prophezeit, eine unglaublich große Speicherdatenbank für Cutting-Down benötigen zu werden. Allerdings greift dieses Lemma auf den Worst-Case eines vollständigen Graphen zurück. In der Realität liegt tatsächlich meistens ein planarer Graph vor, der bis auf einen Innenstadtteil sogar kreisfrei ist. Wäre er gar ein Baum, so wären die Wege eindeutig und wir hätten maximal $\frac{n(n-1)}{2}$ Linien insgesamt in G. Das ist natürlich deutlich besser.

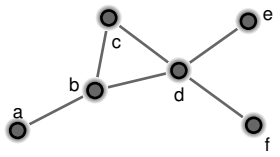
Lemma: 7.2.2 *Sei $G = (V, E)$ ein zusammenhängender Graph mit $|V| = n$. Dann ist die Anzahl aller möglichen Linien in G mindestens $\frac{n(n-1)}{2}$.*

Man könnte auf den Gedanken kommen, dass es umso mehr mögliche Linien gibt, je mehr Kreise im Graphen existieren. Also setzen wir noch einmal an und formulieren ein etwas besseres

Lemma: 7.2.3 *Sei $G = (V, E)$ ein ungerichteter zusammenhängender Graph mit $k \in \mathbb{N}_0$ Kreisen. Dann gibt es insgesamt höchstens $n(n-1)2^{k-1}$ kreisfreie Linien in G.*

Beweis: Hauptcharacteristica einer Linie sind Anfangs- und Endpunkte, die nicht gleich sein dürfen. Das sind wie oben auch schon beschrieben $\frac{n(n-1)}{2}$ Möglichkeiten, die nur noch durch die unterschiedlichen Wegstrecken unterschieden werden können. Da nur nach den kreisfreien Linien gefragt ist, kann keine Linie einen der k Kreise ganz durchlaufen. Wenn eine Linie also einen Kreis durchläuft, hat sie für jeden Kreis höchstens zwei Möglichkeiten, den Kreis zu durchlaufen, wobei wir den Rest der Linie festhalten und nur die Linie in Abhängigkeit des einen Kreises betrachten. Da das Netzwerk nur k Kreise hat, gibt es also maximal $\frac{n(n-1)}{2}2^k = n(n-1)2^{k-1}$ mögliche kreisfreie Linien in G.
Sonderfall für $k = 0$: Bei $k = 0$ handelt es sich bei dem Netzwerk um einen Baum, in dem die Wege eindeutig sind, weswegen es genau $\frac{n(n-1)}{2} = n(n-1)2^{-1} = n(n-1)2^{k-1}$ kreisfreie Wege gibt. □

Beispiel:



In diesem Graph haben wir sechs Knoten und einen Kreis. Nach unserer Abschätzung des oberen Lemmas ergibt das maximal $6(6-1)2^{1-1} = 30$ kreisfreie Linien. In Wahrheit sind es genau 26 kreisfreie Wege. In diesem Beispiel ist die Abschätzung also schon recht nah an der Wirklichkeit.

Dieses Lemma ist auch noch nicht sonderlich gut, weil die obere Schranke für den maximalen Linienpool immer noch exponentiell ist.
In der Praxis werden wir allerdings kaum alle möglichen Linien herausfinden und in unseren Linienpool hinzunehmen. Statt dessen verwenden wir folgende

Definition: 7.2.4 *Sei ein Graph $G = (V, E)$ gegeben. $lines(x, G)$ bezeichnet alle Linien mit x direkten Umwegen. Also $lines(0, G)$ sind alle direkten Wege und $lines(1, G)$ sind alle Wege mit einen Umweg über einen anderen Knoten.*

Lemma: 7.2.5 *Für jedes $x \in \mathbb{N}$ und jeden Graphen $G = (V, E)$ ist $|lines(x, G)| < |V|^{x+2}$.*

Beweis: Man kann die Menge aller Linien also schreiben als einen x+2 Vektor mit den Eckknoten der Linien. Also $lines(1, G) \subset \{(a, b, c)\ s.d.\ a, b, c \in V\}$. Dann besteht die Linie aus dem direkten Weg von a nach b und anschließend vom direkten Weg von b nach c. Analog mit einem anderen x. Dabei repräsentiert (a,a,b) als Vektor die gleiche Linie repräsentiert wie (b,a,a).
Dann gilt natürlich $lines(x, g) \subset lines(y, G)\ \forall x < y$. Und es gilt $|lines(x, g)| \leq |\{(a, b, ..., x + 2)\ s.d.\ a, b, ..., x + 2 \in V\}| = |V|^{x+2}$. □

Lemma: 7.2.6 *Sei $G = (V, E)$ ein PTN mit $|V| = n$. Dann beinhaltet $lines(n - 2, G)$ alle möglichen Linie in G.*

Beweis: Jede Linie in $lines(n - 2, G)$ hat n Eckknoten und damit sind alle kreisfreien Linien möglich. □

In der Praxis wird man für Cutting-Down etwa $lines(1, G)$ oder $lines(2, G)$ verwenden. Wenn man das x fest lässt, so hat man einen überschaubaren Linienpool, der polynomiale Größe hat. Das nette an Cutting-Down ist, dass der Linienpool stets kleiner wird und das Problem damit nach jedem Schritt leichter zu lösen ist.

8 Cluster in Graphen

In diesem Kapitel wollen wir zu einem Graphen seine zugehörigen Cluster, das sind spezielle Subgraphen, definieren und beobachten. Wir werden dabei feststellen, dass man unsere Algorithmen auf Cluster aufteilen kann, also auf kleinere Mengen anwenden kann als unser Ursprungsgraph es zulässt. Wo wir vorher gsagt haben, dass unsere Algorithmen umso besser werden, je weniger Kreise es in einem Netzwerk gibt, sagen wir zukünftig, die Algorithmen werden umso besser, je mehr Cluster wir finden können.
Cluster soll man sich dabei so vorstellen wie Stadtbezirke oder eigene Städte, die bis auf eine Kante vom Rest des Graphen getrennt sind.

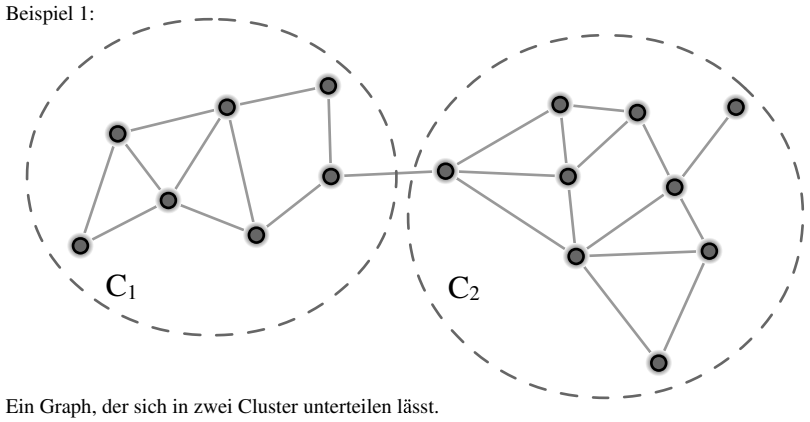
8.1 Definition und Beispiele

Definition: 8.1.1 *Sei $G = (V, E)$ ein Graph mit $|V| = n$ Knoten. G heißt in Cluster unterteilbar, wenn es disjunkte Mengen $C_1 \cup ... \cup C_r = V$ gibt mit $1 < |C_i| < n$ für alle $i = 1, ..., r$, sodass folgendes gilt:*
Für alle $i \neq j$ mit $i, j \in \{1, ..., r\}$ gelte stets

$$\#\{x \in C_i \text{ sodass } \exists y \in C_j \text{ mit } \{x, y\} \in E\} \leq 1$$

Man nennt dann $(C_1, E|_{C_1}), ..., (C_r, E|_{C_r})$ Cluster von G.

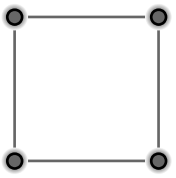
Anschaulich bedeutet diese Definition, dass wir unseren Graphen in Cluster zerlegen können, die zwar ihrerseits Kreise beinhalten können, aber dennoch zwischen keinen zwei Clustern ein direkter Kreisschluss möglich ist. Damit das nicht möglich ist, dürfen keine zwei Cluster zwei oder mehr sie trennende Kanten haben.



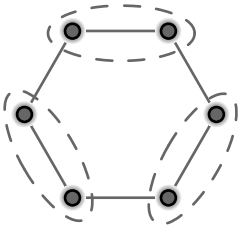
Ein Graph, der sich in zwei Cluster unterteilen lässt.

Überdies kann es durchaus sein, dass ein Graph sich in Cluster unterteilen lässt, obwohl er Kreisstruktur besitzt, wie ein einfaches Beipiel zeigt:

Beispiel 2:



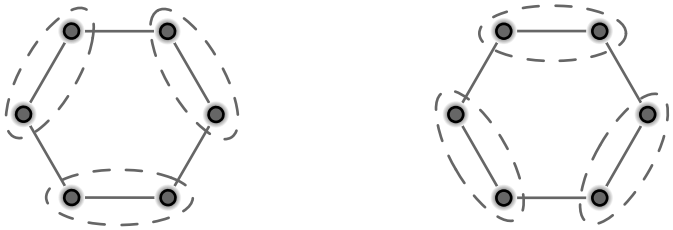
Die Quadratur des Kreises: Wie man es drehen und wenden mag - dieser "Kreis" lässt sich nicht in Cluster zerlegen.



Hier dagegen ein Kreis, der sich sehr wohl in Cluster zerfällt.

Bemerkung: Die Aufteilung eines Graphen in Cluster ist nicht eindeutig. Das kann man sehr leicht sehen, wenn man sich einen einfachen Kreis mit sechs Knoten anschaut (siehe oben) und in drei Cluster à zwei Knoten aufteilt. Das ist eine gültige Aufteilung in Cluster, weil jeder Cluster zwei Nachbarcluster hat, zu dem es nur eine Verbindungskante gibt. Aber es gibt zwei verschiedene Unterteilungen, weil die Cluster auch um einen Knoten versetzt definiert werden könnten (siehe Abbildung).

Beispiel 3:



Zwei verschiedene Zerlegungen in Cluster bei ein und demselben Graphen.

Lemma: 8.1.2 *Kein vollständiger Graph ist in Cluster unterteilbar.*

Beweis: Zuerst zeigen wir, dass jeder Graph, der in Cluster unterteilbar ist, mindestens vier Knoten besitzen muss. Weil $1 < |C_i|$ für alle $i = 1, ..., r$ gilt und alle C_i disjunkt sein müssen, haben wir $|C_i| \geq 2$. Weil nun $|C_i| < n$, aber $C_1 \cup ... \cup C_r = V$ gelten muss, ist $r \geq 2$. Daraus gilt für ein minimales Netz G, das in Cluster unterteilbar ist, dass $n = |C_1| + |C_2| = 2 + 2 = 4$.
Jetzt nehmen wir an, dass der vollständige Graph G_n mit $n \geq 4$ Knoten zerlegbar in Cluster ist und führen das zu einem Widerspruch. Sei C_1 ein Cluster. Weil $1 < |C_1| < n$, hat er mindestens zwei Knoten, ist aber auch nicht der einzige Cluster. Demnach existiert ein Cluster, das wir oEdA C_2 nennen. Jeder der mindestens zwei Knoten von C_1 hat je eine Kante in E zu einem der mindestens zwei Knoten von C_2, was also $\#\{x \in C_1 \; sodass \; \exists y \in C_2 \; mit \; \{x, y\} \in E\} \geq 4$ ergibt, was ein Widerspruch dazu ist, dass G_n sich in Cluster unterteilen lässt. ↯
Damit kann es keinen vollständigen Graphen mit vier oder mehr Knoten geben, der in Cluster unterteilbar ist. Vollständige Graphen mit weniger als vier Knoten sind nicht in Cluster unterteilbar, weil kein Graph mit weniger als vier Knoten in Cluster zerlegbar ist, wie oben gezeigt wurde. □

Lemma: 8.1.3 *Sei $G = (V, E)$ ein zusammenhängender und kreisfreier Graph und $C_1 \cup C_2 = V$ disjunkt und selbst zusammenhängend mit $|C_1|, |C_2| \geq 2$. Dann ist G in Cluster unterteilbar und $(C_1, E|_{C_1})$ und $(C_2, E|_{C_2})$ sind Cluster von G.*

Beweis: Angenommen, es gebe zwischen C_1 und C_2 zwei oder mehr Kanten, dann würde wegen des Zusammenhangs von C_1 und C_2 ein Kreis in G existieren. $\lightning$
Also gilt $\#\{x \in C_1 \; sodass \; \exists y \in C_2 \; mit \; \{x, y\} \in E\} \leq 1$, weswegen G in Cluster unterteilbar ist. □

Im Gegensatz zu vollständigen Graphen kann man also Bäume nicht nur einfach unterteilen in Cluster; man kann sie sogar fast beliebig in Cluster unterteilen. Da haben wir also wieder unsere zwei Extrema: Vollständige Graphen und Bäume. Und das ganze Feld dazwischen versuchen wir nun nicht mehr mit der Anzahl der Kreise (wie in Kapitel 6) sondern mittels Cluster einzuordnen.

Definition: 8.1.4 *Sei $G = (V, E)$ ein Graph und $(C_1, E|_{C_1}), ..., (C_r, E|_{C_r})$ eine Clusterunterteilung. Dann bezeichne $(\{C_1, ..., C_r\}, E \backslash (E|_{C_1} \cup ... \cup E|_{C_r})$ einen Clustergraphen von G.*

Die Bedeutung des Clustergraphen ergibt sich daraus, dass man ein Problem des ganzen Netzwerks auf den wesentlich kleineren Clustergraphen und den verschiedenen Clustern lösen kann.

8.2 Zerlegung eines Problems in Subprobleme mittels Cluster

Nun wollen wir das Thema etwas konkreter machen und unsere Erkenntnisse über Cluster auf die PTNs und Linienprobleme übertragen.

Lemma: 8.2.1 *Sei ein PTN $G = (V, E)$ gegeben, das in Cluster unterteilbar ist. Gesucht soll das serielle Linienproblem ohne Budget sein. Ist der Clustergraph kreisfrei, so kann man eine Lösung für G konstruieren aus den optimalen Lösungen der Cluster.*

Beweis: Wenn der Clustergraph kreisfrei ist, sind die Wege der OD-Paare durch den Clustergraph eindeutig. Daraus folgt, dass man mit Split&Merge eine optimale Lösung für den Clustergraph finden kann. Im Folgenden kann man Optimallösungen für die Cluster finden, indem man berücksichtigt, dass OD-Paare anderer Cluster, die im Clustergraphen den zu lösenden Cluster durchlaufen, in diesem Cluster auch als OD-Paare auftauchen.
Durch das Lösen der Cluster wird die Lösung des Clustergraphen nicht un-optimal, weil die Wege im Clustergraphen eindeutig sind. Also ergeben alle Lösungen zusammen eine optimale Lösung für ganz G. □

Dies gilt vorerst nur für kreifreie Clustergraphen. Bei Clustergraphen, die Kreise besitzen, kann man nicht von der Eindeutigkeit der Wege ausgehen. Um das zu umgehen, kann man sowas wie den Schritt vom H_0-Algorithmus zum H-Algorithmus um ein weiteres Mal vollziehen.
Im Grunde funktioniert dieser Algorithmus auch für das parallele Linienproblem mit Budget. Man muss nur das Lösen des Clustergraphen mit Split&Merge mit fester Kantenkardinalität tätigen. Das Lösen der Cluster kann man dann natürlich nicht mehr optimal schaffen (zumindest nicht mit den hier vorgestellten Algorithmen), und es ist egal bzw. Geschmackssache, ob zum Lösen der Cluster der Split&Merge oder der Cutting-Down Algorithmus verwendet wird.

9 Gemischte Verkehrsnetze

In vielen Städten und besonders in denjenigen Städten, die am meisten Haltestellen und damit Knoten in den zugehörigen PTNs besitzen, haben Passagiere des Nahverkehrsnetzes nicht nur die Möglichkeit, sich mit Bussen, sondern auch mit Straßen- und Untergrundbahnen oder anderen Fortbewegungsmöglichkeiten zu bewegen. Gerade wenn man auf Reisezeiten und Umsteigezahlen achten und diese für die Passagiere optimieren will, muss man zusätzliches Augenmerk darauf legen, dass viele Reisegäste häufig aus einer U-Bahn aussteigen und dann mit dem Bus weiter reisen wollen. Ändert sich dann das Busnetz, wie wir es in den vorgestellten Algorithmen häufig zumindest virtuell passiert, so würde der Fahrgast oftmals auch eine andere U-Bahn nehmen, um danach in eine andere Buslinie als zuvor umzusteigen. Diese Überlegung bedeutet im Grunde nichts anderes, als dass man, um dies mit zu berücksichtigen, die ganzen anderen Nahverkehrsnetze mit in unser PTN einbeziehen muss und für die Passagiere stets als zusätzliche Umsteigemöglichkeit anzubieten.

Das ist allerdings dank des Change&Go-Graphen gar kein Problem. Wir fügen das feste Verkehrsnetz zum Change&Go-Graphen hinzu, beziehen es allerdings nicht in unsere Algorithmen ein. Das funktioniert sowohl für den Split&Merge als auch für den Cutting-Down Algorithmus.

9.1 Cutting-Down Heuristik mit festem Verkehrsnetz

Input: PTN $G = (V, E)$ mit OD-Matrix W, einem Budget $b \in \mathbb{R}$ und ein Linienpool F eines festen Verkehrsnetzes, das nicht verändert werden kann.

1. Erzeuge kompletten Linienpool L mit allen kreisfreien Linien und einfachen Kreisen, sowie eine Indexmenge $I \subset \mathbb{N}$ von L.
2. Erzeuge Change&Go Graphen G' aus $L \cup F$.
3. Solange $c(L) > b$ führe aus:
   (a) Für alle $l_i \in L$ bestimme $k_i := \sum_{p \in P} t(p, L \setminus \{l_i\})$.
   (b) Bestimme $\min_I\{k_i\}$ und streiche entsprechend $L := L \setminus \{l_i\}$, sowie $I := I \setminus \{i\}$, sowie l_i aus G'.

Output: Linienverbund L.

9.2 Split & Merge Algorithmus mit Budgetgrenze und festem Verkehrsnetz

Input: Ein PTN G mit der OD-Matrix W, einem Budget $b \in \mathbb{R}$, sowie einer Kostenfunktion $c : \mathcal{L}_G \to \mathbb{R}$ und einem Linienpool F, der nicht verändert werden darf.

1. $L = \varnothing$; Berechne minimalen spannender Baum T; $\forall e \in T$ füge Weg (e) zu L hinzu.
2. Initialisiere den Change&Go Graph $G' = (V', E')$ aus $L \cup F$ und die n Matchingprobleme $M_1, ..., M_n$ aus L und $time = \infty$.
3. Solange $zfw\_alt \neq \sum_{p \in P} t(p, L)$ tu

   (a) $zfw\_alt = \sum_{p \in P} t(p, L)$.
   (b) Für alle Knoten $v_i \in V$ ermittle L neu durch Subalgorithmus 2 an v_i.
   (c) Wenn $c(L) < b$ erweitere L um maximal $b - c(L)$.

4. Output: Linienverbund L.

10 Die Algorithmen in der Anwendung

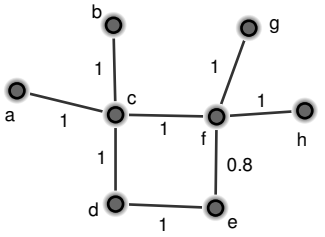
Jetzt wollen wir uns anschauen, inwieweit unseren netten theoretischen Überlegungen für die Umsetzung taugen. Dazu wurden Split&Merge und Cutting-Down jeweils in C++ programmiert. Dieses Programm hier en detail abzudrucken, ist mit seinen gut 1700 Code-Zeilen sicherlich zu viel. Aber ein paar Rahmendaten sollen doch vorgestellt werden.

10.1 Das Programm in der Umsetzung

Es handelt sich um ein in C++ geschriebenes Kommandozeilenprogramm, das auf allen gängigen Betriebssystemen läuft. Zum Kompilieren wurden nur die gcc Standard-Bibliotheken verwendet, also keine ausgefallenen Zusatzbibliotheken wie Schnittstellen zu Matlab oder XPress-MP.
Programmiert wurden konkret der Split&Merge Algorithmus einmal mit vorgegebenem Pool und dann noch einmal mit Budget. Zudem wurde der Cutting-Down Algorithmus implementiert, der nur mit einem Budget arbeitet.
Das Programm wurde so designed, dass es sich nahtlos in das „LinTim“ (Lineplanning & Timetabling) Projekt von Professor Schöbel an der Universität Göttingen einfügt. Das bedeutet konkret, dass es das Eingabenetzwerk und etwaige Pools in einem untergeordneten Verzeichnis in einem einheitlichen Dateienformat erhält. Dadurch ist es sehr flexibel in Kombination mit anderen Algorithmen einsetzbar.
Der Referrenz-Rechner ist ein AMD Athlon 6000+ auf 3 GHz getaktet mit 2 GB RAM, also ein durchaus im Jahr 2008 gängiger Privatrechner. Der Rechner verfügt zwar über eine Doppelkern-CPU, das Programm ist aber nicht für parallel processing ausgelegt und konnte daher nur einen Kern sinnvoll einsetzen. Es ist also sowohl in der Rechnerleistung als auch in der Programmierung gewissermaßen noch Luft nach oben.
Aus eben jenem Lintim-Projekt stammt auch das erste kleine Beispiel, auf das wir das Programm anwenden wollen.

10.2 Das Spiel-Beispiel

Es handelt sich dabei um ein Beispiel, das so klein gehalten wurde, dass man noch die Übersicht darüber behalten kann. Nur acht Haltestellen umfasst das fiktive Verkehrsnetzwerk und ebensoviele Kanten.



$$W = \begin{pmatrix} 0 & & & & & & & \\ 10 & 0 & & & & & & \\ 20 & 40 & 0 & & & & & \\ 10 & 20 & 20 & 0 & & & & \\ 60 & 70 & 100 & 100 & 0 & & & \\ 100 & 100 & 50 & 150 & 120 & 0 & & \\ 0 & 0 & 20 & 0 & 100 & 60 & 0 & \\ 0 & 10 & 10 & 0 & 90 & 50 & 0 & 0 \end{pmatrix}$$

Die Knoten sind wie auf dem obigen Schaubild mit ungerichteten Kanten verbunden, die mit ihrer Länge beschriftet worden sind. Also haben alle Kanten eine Länge von 1 bis auf die zwischen e und f, die Länge 0.8 hat. W ist unsere OD-Matrix, die wir als untere Dreiecksmatrix geschrieben haben, weil es in einem ungerichteten Graphen eh keinen Unterschied macht, in welche Richtung die Fahrgäste wollen.

Zudem gibt es schon ein Liniennetz auf unserem Graphen mit acht Linien - hier beschrieben über die Knoten und nicht wie gewöhnlich über die Kanten, um zusätzliche

Beschriftungen zu vermeiden:

$$L = \left\{ \begin{array}{c} (a,c,f,g), \\ (b,c,f,h), \\ (e,f), \\ (c,f,e), \\ (b,c,d), \\ (d,e,f,c), \\ (c,d,e,f), \\ (a,c,d,e,f,g) \end{array} \right\}$$

Diese Linien kosten zusammen ein Budget von 29, wenn die Kostenfunktion für jede Linie 1 plus die Weglänge der Linie ist. Also gilt auch $c(L) = 29$.
Jeden Umsteigevorgang der Fahrgäste bestrafen wir (natürlich nur im Sinne der Fahrgäste) mit 4.
Dies alles zusammen inklusive OD-Matrix, Linienverbund L, der Kostenfunktion c, sowie der Bestrafungszeit von 4 pro Umsteigevorgang ist unser Spiel-Beispiel, auf das wir auch später noch zurückgreifen werden.

Ab jetzt tritt unser Programm in Aktion. Schon die Zielfunktion $\sum_{p \ P} t(p, L)$ für den in LinTim angegebenen Linienpool L auszurechnen, übersteigt den Platz einer DIN A4 Seite. Kein Problem: unser Programm gibt uns aus, dass die Fahrgäste alle zusammen 2854 (Zeit bzw. Weg inklusive Bestrafung) brauchen. Also gilt natürlich auch: $\sum_{p \ P} t(p, L) = 2854$.
Unser Programm gibt uns auch eine Auskunft darüber, dass die Fahrgäste bei maximalem Linienpool (also auf direktem Weg und ohne Umsteigen) zusammen 2162 brauchen, was eine untere Schranke für unser paralleles Linienproblem mit Budget ist. Also können wir uns maximal um 692 verbessern - ganz egal, was wir tun. Es ist aber nicht gesagt, dass wir mit einem Budget die 2162 komplett erreichen können.

Split & Merge mit vorgegebenen Linienverbund

Zuerst wenden wir Split&Merge auf das Spiel-Beispiel an, ohne die Kantenkardinalität zu verändern. Wir schneiden lediglich die bestehenden Linien auf und entsprechend der Fahrgastinformationen wieder zusammen. Wir erhalten nach zwei vollen Durchläufen, wovon sich nur im ersten etwas verbessert hat, einen neuen Verbund (wieder in Knotenschreibweise):

$$L' = \left\{ \begin{array}{c} (a,c,f,g), \\ (a,c,d,e,f,g) \\ (d,e,f,c,f) \\ (b,c,f,e,f) \\ (c,d,e,f,h) \\ (b,c,d) \end{array} \right\}$$

Dieser neue Verbund hat Zielfunktionswert $\sum_{p \ P} t(p, L') = 2272$, was schon 582 Minuten besser ist als zuvor und nur noch 110 von der unteren Schranke entfernt.
Man sieht auch, dass L' kompakter ist; es gibt eine Linie weniger, weswegen sich die Kosten um 1 auf $c(L') = 28$ verbessert haben. Dafür existieren weniger kurze Linien - Split&Merge scheint lange Linien zu mögen, was ja auch durchaus Sinn ergibt, weil bei kurzen Linien tendenziell mehr Fahrgäste und manche Fahrgäste auch häufiger umsteigen müssen. Das ist doppelt gut, denn lange Linien kosten in der Summe weniger.

Algorithmen mit Budget

Und jetzt versuchen wir, ganz ohne anfänglichen Linienverbund eine besonders gute Lösung zu suchen. Die Algorithmen, die wir dafür verwenden, sind Split&Merge und Cutting-Down.
Bei Split&Merge fangen wir stets von einem MST aus an und kaufen uns in jedem Schritt neue Kanten hinzu. Da dies, wie bereits erwähnt, ein schwieriges Problem für sich ist, wenden wir einfach mehrere Möglichkeiten, sich neue Kanten hinzuzukaufen, an und vergleichen sie. Im Konkreten sind diese Zukaufsmöglichkeiten:

- **OD-Paar:** Erweitern durch jeweils schlechtestes OD-Paar, also das OD-Paar, das am meisten Umsteigevorgänge mal Anzahl Fahrgästen vorzuweisen hat. Von diesem Verfahren erwarten wir gute Ergebnisse, wenn das Budget groß ist, und schlechte, wenn das Budget nicht ausreicht. Es ist also kein Vorgehen, das in der Lage ist, Kompromisse in der Linienwahl zu treffen, wie ja Split&Merge eigentlich gedacht ist. Die beiden Teilalgorithmen werden also wenig wahrscheinlich gut harmonieren.
- **Greedy:** Es wird von allen direkten Linien (also Linien, die keine Umwege fahren) diejenige genommen, die zusammen mit dem bisherigen Verbund und nach einem anschließenden Split&Merge Schritt den besten Zielfunktionswert hat.
- **Greedy mit größerer Wirtschaftlichkeit** bzw. Greedy 2: Dieses Mal arbeiten wir wie oben beim Greedy, nur dass die Linie zugekauft wird, deren Verbesserung des Zielfunktionswertes durch ihre Länge am größten ist. Es wird also dieses Mal die Länge einer Linie mit berücksichtigt und verstärkt kürzere Linien eingekauft. Davon erhoffen wir uns, dass mehr Budget übrig bleibt, im Endeffekt also mehr Linien eingekauft werden können.

Der Cutting-Down Algorithmus hängt sehr stark von der Reihenfolge der Linien in der Liste aller Linien, die er am Anfang als Input übergeben bekommt, ab. Deswegen wollen wir Cutting-Down stochastisch bereinigt zehn mal mit zufällig vertauschter Listenreihenfolge ausführen und einen Mittelwert abgeben. Auf dieselbe Weise wird auch die Liste aller direkten Linien bei den beiden Split&Merges mit Greedy von der Stochastik bereinigt, auch wenn hier die Abhängigkeit des Zufalls nicht so groß sein mag wie bei Cutting-Down.
Weil das Budget eine Rolle spielen wird, werden wir auch verschiedene Budgetgrenzen angeben.
Die Ergebnisse werden in der folgenden Tabelle dargestellt:

Budget	$S\&M_{OD}$	$S\&M_{Greedy}$	$S\&M_{Greedy2}$	Cutting-Down
29	2558	2294	2238	2222
28	2958	2342	2238	2232
27	2958	2334	2318	2235,4
26	2958	2366	2318	2225,6
25	3038	2382	2318	2253.2
24	3038	2398	2318	2269,6
23	3118	2445	2398	2327,4
22	3118	2492	2398	2351,4
21	3118	2492	2398	2366,4
20	3118	2630	2492	$2413,1 + \infty$
19	3758	2852	2852	2368,2
18	3758	2852	2852	$2506 + 5\infty$
17	3758	3012	2852	∞
16	3758	3252	3252	$2450 + 9\infty$
15	3852	3252	3252	∞

Was sofort auffällt ist, dass S&M Greedy2 stets besser oder zumindest gleich ist wie S&M Greedy, was nicht gerade einsehbar (also nicht mathematisch trivial beweisbar) ist. Zudem hat sich unsere Vermutung bestätigt, dass S&M OD ein ziemlich schlechter Algorithmus ist.
Insgesammt die besten Ergebnisse bringt uns der Cutting-Down Algorithmus, der mit 36 Sekunden im Gegensatz zu höchstens 0,2 Sekunden der S&M-Algorithmen deutlich am langsamsten ist. Die Ergebnisse des Cutting-Down in der Tabelle sind Durchschnittswerte von jeweils zehn Zählungen. Im Ergebnis hat Cutting-Down zwar einige schlechtere, aber auch einige sehr gute Lösungen inklusive vereinzelter Optimallösungen gefunden. Wenn man also Zeit hat, könnte sich der Cutting-Down Algorithmus am ehesten lohnen.

Vergleich mit anderen Algorithmen

Es gibt zwar in der Literatur keine Algorithmen der Linienplanung, die unser Problem konkret behandeln, und auch nicht den gleichen Input haben. Aber es gibt Algorithmen, die einen aehnlichen Output haben. Im LinTim-Projekt sind konkret implementiert die Algorithmen H6 und H7 (aus Optimization Models in Public Transportation [7]). Sie benutzen stets einen Linienpool und wählen aus diesem anhand von maximalen und minimalen Flussgrenzen einige Linien und ihre Frequenzen aus, die sie als fertiges Linienkonzept ausgeben. Einerseits liefern sie damit mehr als unsere erarbeiteten Algorithmen, weil sie statt eines Linienverbundes ein Linienkonzept ausgeben. Aber in diesem Linienkonzept steckt auch ein Linienverbund, den wir mit den von unseren Algorithmen erzeugten Linienverbünden vergleichen können.
Output von H6: 3150 bei Budget 13,6

$$L_{H6} = \left\{ \begin{matrix} (b,c,f,h), \\ (d,e,f,c) \\ (a,c,d,e,f,g) \end{matrix} \right\}$$

Output von H7: 2854 bei Budget 25,2

$$L_{H7} = \left\{ \begin{array}{c} (a,c,f,g), \\ (b,c,f,h) \\ (e,f) \\ (c,f,e) \\ (b,c,d) \\ (d,e,f,c) \\ (a,c,d,e,f,g) \end{array} \right\}$$

Man erkennt an diesen beiden Outputs, dass die Ergebnisse deutlich unterschiedlich sind, obwohl die Eingabedaten dieselben sind. Beide Ergebnisse sind auch in unserem Sinne zulässig, was nicht verwundert, da unser Begriff der Zulässigkeit Äquivalent zur Zulässigkeit ist, die [7] verwendet, nämlich, dass alle Fahrgäste an ihr Ziel kommen können müssen.
Das Ergebnis von H6 ist gar nicht schlecht im Vergleich zu anderen Linienverbünden mit ähnlichem Budget. Es stellt sich die Frage, warum ein so geringes Budget verwendet worden ist. Tatsächlich aber ist ein besonders großes Budget verwendet worden. In seiner eigenen Kostenfunktion nämlich fließt auch die Frequenz einer Linie mit ein. Die Frequenzen der drei Linien sind im Linienkonzept nun aber 3, 7 und 9, was nicht gerade wenig ist. Heraus kommt ein Budget von 94, was ziemlich viel ist, wenn an bedenkt, dass der Output von H7 mit mehr als doppelt so vielen Linien gerade einmal 50 beträgt (beide Male die eigene Kostenfunktion verwendet). Also tatsächlich ist der Output von H6 als Linienkonzept ziemlich schlecht.
Der Output von H7 ist deutlich günstiger, aber unsere Bewertung des dazu passenden Linienverbundes ist nicht gerade gut im Vergleich. Bei einem Budget von 25,2 (nach unserer Kostenfunktion) könnnen unsere Algorithmen mit Ausnahme von Split&Merge mit OD-Erweiterung deutlich bessere Ergebnisse liefern. 2318 statt 2854 ist deutlich näher an der oberen Schranke von 2162.

Jetzt stellt sich ganz automatisch die Frage, was passiert, wenn die Algorithmen H6 und H7 für Ergebnisse bekommen, wenn wir ihnen einen von unseren Algorithmen erzeugten Linienverbund als Input geben. Alles, was die Algorithmen als zusätzliche Input-Informationen brauchen, sind Load-Ober- und Untergrenzen der Kanten, also wieviele Fahrgäste höchstens bzw. mindestens über eine Kante fahren müssen, und die Kosten der einzelnen Linien, die im Verbund bzw. Pool sind. Das ist aber ist einfach zu bewerkstelligen. Wir geben als Input einen sehr guten Linienverbund mit Budget 26,4, bei dem kein Fahrgast umsteigen muss, sondern höchstens kleine Umwege fährt. In diesem Linienverbund sind 5 Linien. Und die Ergebnisse sind:

Output von H6:

$$L_{H6} = \left\{ \begin{array}{c} (a,c,b,c,f,h), \\ (a,c,d,e,f,h) \\ (a,c,f,g,f,e) \end{array} \right\}$$

mit Frequenzen 7, 3 und 9 und internen Kosten des Konzeptes von 102.
Output von H7:

$$L_{H7} = \left\{ \begin{array}{c} (b,c,f,e), \\ (a,c,d,e,f,h) \\ (a,c,f,g,f,e) \end{array} \right\}$$

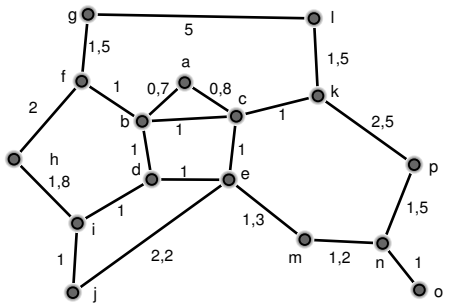
mit Frequenzen 5, 3 und 2 und internen Kosten des Konzeptes von 40.

Das sind etwas durchwachsende Ergebnisse. Das Ergebnis von H6 scheint komplett Blödsinn zu sein. Dass ein Bus 9 mal in der Stunde fährt, ist kein besonders schönes Resultat. H7 bietet da schon einen deutlich schöneren Output mit verträglichen Frequenzen. Aber auch hier wird der gemäß unserer eigenen Zielfunktion sehr gute Linienverbund nicht in seiner Struktur erfasst, sondern wieder zerstückelt. Trotzdem ist zumindest H7 in der Lage, mit unserem guten Linienverbund eine kostengüstigere Lösung zu erzeugen als mit dem vorgegebenen Pool.

Eine wichtige Frage ist: wie können H6 und H7 so gute Ergebnisse für unseren Zielfunktionswert bekommen, wo sie sich um unsere Zielfunktion eigentlich gar nicht kümmern? Einerseits kann das sein, weil das Beispiel mit 8 Knoten und nur einem Kreis sehr klein ist und daher die kürzesten Wege im PTN in etwa den echten Wegen im Change&Go Graphenm entsprechen. Zum anderen kann es daran liegen, dass der vorgegebene Pool, den H6 und H7 benötigen, in diesem Beispiel sich ganz gut eignet für die Verwendung von H6 und H7. Die Linien haben die richtige Länge und es sind nicht zu viele davon. Würden beispielsweise alle Linien von $lines(2, G)$ als Input gegeben sein, so würde das Ergebnis vielleicht nicht ganz so gut aussehen. Oder mit anderen Worten: Es ist schon eine Menge Vorarbeit geleistet worden, bevor H6 und H7 überhaupt arbeiten konnten.

10.3 Das Spiel16 Beispiel

Nicht Teil des LinTim-Projektes ist das folgende Beispiel. Es wurde von mir selbst entworfen, aber in Anlehnung an das eben gezeigte Spiel-Beispiel. Die Knotenanzahl wurde von 8 auf 16 erhöht, die Kantenlänge ist in etwa gleich geblieben, ebenso auch die Kosten einer Linie und die Umsteigezeit.



Die OD-Matrix soll hier nicht weiter aufgeführt werden. Aber es gibt tendenziell ähnlich viele Fahrgäste als einzelnes OD-Paar wie beim Spiel-Beispiel und wieder fahren wenig Fahrgäste über sehr lange und sehr kurze Distanzen und am meisten über die Mitteldistanzen.

Split & Merge mit vorgegebenen Pool

Es gibt keinen vorgegebenen Pool. Deswegen kann er auch nicht optimiert werden. Wir betrachten das Spiel16 Beispiel einfach als eine Anwendung auf eine vollkommen neue Stadt, die jetzt völlig neu ein Busnetz bekommen soll. Deswegen gehen wir gleich über zu den ...

Algorithmen mit Budget

Weil das PTN doppelt so viele Knoten wie das eigentliche Spiel-Beispiel hat, starten wir jetzt auch mit einem doppelt so großen Budget. Das Budget von 30 für das

Spiel-Beispiel war völlig überzogen und deswegen gehen wir einfach davon aus, dass ein Budget von 60 für Spiel16 nicht gerade wenig sein wird. Wieder verringern wir das Budget schrittweise.
Weil Cutting-Down mit $lines(2, G)$ schon für dieses relativ kleine Beispiel viel zu lange dauern würde, wenden wir Cutting-Down mit $lines(1, G)$ an. Das verfälscht leider unser Ergebnis ein bisschen, wie wir gleich sehen werden:

Budget	$S\&M_{OD}$	$S\&M_{Greedy}$	$S\&M_{Greedy2}$	Cutting-Down
60	12937,5	11865,5	11437,5	11484,6
55	13329,5	12525,5	12303,5	11828,7
50	13769,5	13237,5	12303,5	12339,1
45	13841,5	13410,3	12703,5	13093,5
40	13841,5	14101,5	13415,5	13525,5
35	14707,5	15425,5	14247,5	13933,5
30	16014	17902	15554	15203
25	17902	17902	17902	16911

Sofort fällt auf, dass Split&Merge mit Greedy2 Erweiterung, also der wirtschaftlichen Greedy-Erweiterung, sich ein Kopf-an-Kopf Rennen mit Cutting-Down liefert. Dass Cutting-Down nicht mehr so deutlich besser ist als seine Konkurrenten, mag daran liegen, dass er mit $lines(1, G)$ einen deutlich kleineren Anfangs-Pool verwendet. Erstaunlich ist dann aber doch, dass Cutting-Down gerade bei kleinem Budget besonders gut wird.

10.4 Das Bahn-Klein Beispiel

In diesem Fall handelt es sich um ein realistisches Netz, das allerdings nicht ein Bussondern ein Bahn-Netzwerk darstellt. Entnommen ist es wiederum dem LinTim Projekt. Die Daten sind leicht abgefälscht, aber trotzdem so, dass die Ergebnisse realistisch bleiben.
Auch wenn es sich um ein Bahn-Netzwerk handelt, so kann man unsere Algorithmen problemlos anwenden, weil wir auch bei der Bahn lediglich Linien haben und Fahrgäste, die von Linie zu Linie umsteigen müssen.
Das PTN ist deutlich größer als unser anderes Beispiel: Es besteht aus exakt 250 Knoten und 326 Kanten. Der ursprüngliche Pool besteht zudem aus 132 Linien. Jede Kante ist so zwischen 1000 und 100000 lang, weswegen wir unsere Umsteigedauer also die Bestrafungszeit auf etwa 50000 setzen, was zugegeben ein sehr improvisierter statt reiflich überlegter Wert ist. Das aber macht nichts, wie sich gleich zeigen wird.

Split & Merge mit vorgegebenen Pool

Wir fangen wieder einmal mit dem Split&Merge mit vorgegebenem Pool an. Der Algorithmus braucht mit 7 Minuten deutlich länger als zuvor, eine sinnvolle Lösung heraus zu finden. Und auch bei den Zielfunktionswerten wird es allmählich unübersichtlich: **Gegebener Pool:** $5,58399 \cdot 10^1 1$.
Optimierter Pool: $5,58399 \cdot 10^1 1$
Also hat sich keine oder zumindest keine prozentual spührbare Verbesserung ergeben. Schauen wir uns die untere Schranke an, wird auch recht schnell klar, woran das liegen muss.
Untere Schranke: $5,08201 \cdot 10^1 1$.
Also liegt der gegebene Pool nur 9,877 % über der unteren Schranke, was nicht sehr viel ist. Es scheint also kaum Fahrgäste zu geben, die wirklich umsteigen müssen. Es

gibt also fast nichts zu optimieren bzw. es ist fraglich, ob überhaupt etwas optimiert werden kann, weil das Budget ja nicht für die untere Schranke ausreichen muss.

Algorithmen mit Budget

Wir wollen wie beim Spiel-Beispiel jetzt einmal die Algorithmen anwenden, die mit Budget arbeiten. Damit man auch etwas sehen kann (und damit die Laufzeit der Algorithmen kleiner wird), gehen wir von einem kleineren Budget aus.
Im Anfangspool (der ja schon fast optimal war), gab es 132 Linien. Wir reduzieren das Budget deutlich, also zuerst auf die etwa die Hälfte, was 50 Mio. entspricht bzw. 75 Linien, und dann noch weiter. Hier die Tabelle der Ergebnisse dazu:

Budget	$S\&M_{OD}$	$S\&M_{Greedy}$	$S\&M_{Greedy2}$	Cutting-Down
50 Mio.	$5,11715 \cdot 10^{11}$	??	??	Zu wenig Speicher
40 Mio.	$5,12596 \cdot 10^{11}$	??	??	Zu wenig Speicher
30 Mio.	$5,12597 \cdot 10^{11}$	??	??	Zu wenig Speicher
23 Mio.	$5,16442 \cdot 10^{11}$	??	??	Zu wenig Speicher
12 Mio.	$6,97761 \cdot 10^{11}$	??	??	Zu wenig Speicher

Die Split&Merge Greedy (1 und 2) Funktionen brauchten im Test zu lange, als dass es hier den Aufwand wert gewesen wäre. Nach sieben Stunden haben sie keinen Schritt geschafft, womit eine Hochrechnung ergibt, dass sie ungefähr mindestens drei Wochen benötigen würden, um zu einem Ergebnis zu gelangen.
Man sieht also recht schnell, dass 250 Knoten zu viel sind, um zumindest auf die Schnelle eine schöne Tabelle an Ergebnissen zustande zu bekommen, aber auch, dass gerade die Erweiterung des Linienpools einen zeitlichen Unterschied machen kann. Für die letzendliche Anwendung ist dies kein KO-Kriterium, weil ja unter Umständen viel Zeit vorhanden ist, um einen Fahrplan auszurechnen. Drei Wochen würden jedenfalls in Ordnung sein. Nebenbei muss gesagt werden, dass Busnetze selten mehr als 250 Haltestellen über die Stadt verteilt haben. Auch ist das Programm hier natürlich noch nicht endgültig laufzeitoptimiert, sodass das Programm bestimmt auch noch etwas schneller werden kann, wenn man mehr Arbeit investiert.
Wir können auch feststellen, dass Greedy-Heuristiken zur Erweiterung des Linienpools sehr langsam werden, weil in einem großen Netzwerk alle Möglichkeiten einmal durchgespielt werden müssen, um zu sehen, welche die Beste ist. Hier ist es mit Sicherheit effektiver, andere Methoden zu entwickeln. In diesen Punkt, die Erweiterung des Pools, haben wir bisher weniger Arbeit investiert.

10.5 Linienverbünde und Verspätungsmanagement

Im Rahmen des LinTim-Projektes gibt es auch die Möglichkeit, integriert Algorithmen der Verkehrsplanung hintereinander ausführen zu können. Zur Zeit sind die Ergebnisse noch sehr jung und frisch und benötigen noch weitere Untersuchungen. Aber verheimlicht sollen sie hier auch nicht werden.
Die Idee des itegrierten Arbeitens mit Algorithmen ist, dass viele Schritte der Verkehrsplanung bisher für sich betrachtet worden sind, aber im Grunde viel miteinander zu tun haben. So kann es keinen Fahrplan geben, wenn man keinen Linienverbund hat. Und ohne Fahrplan wiederum kann man kein Verspätungsmanagment

durchführen, also entscheiden, ob ein abfahrender Zug auf einen zu spät kommenden Zug warten soll oder nicht. In LinTim kann man diese Schritte konkret hintereinander ausführen und am Ende betrachten, inwieweit ein guter Linienverbund sich auf das Verspätungsmanagement auswirkt. Die Hypothese muss hier sein: in einem Linienverbund, wo wenige Fahrgäste überhaupt umsteigen müssen, hat das Verspätungsmanagement wenig zu tun bzw. kann selbst bei schlechten Entscheidungen weniger Schaden zufügen.
Die Kette, die wir uns anschauen, ist also:

$$\textit{Erstellen eines Linienkonzeptes}$$
$$\Downarrow$$
$$\textit{Erzeugung eines Fahrplans}$$
$$\Downarrow$$
$$\textit{Verspätungsmanagement}$$

Um an unser Linienkonzept zu kommen, wird zweimal Split&Merge mit Greedy2-Heuristik angewendet und zweimal Cutting-Down. Zudem verwenden wir noch einmal die Algorithmen H6 und H7 und eine gefundene Optimallösung des Linienkonzeptproblems mit Mosel (mit vorgegebenem Pool).
Der Fahrplan wird leider nur approximativ berechnet, wozu also keine Optimalitätsaussage getroffen werden kann - also gar keine. Und das Verspätungsmanagement wurde als ganzzahliges Problem ([3, 8]) mit stochastisch verteilten Verspätungsszenarien über viele hundert Berechnungen ermittelt.
Die Ergebnisse werden in der folgenden Tabelle dargestellt:

	H6	H7	Mosel	S&M	S&M	C-D	C-D
Linien-ZfW	3150	3070	2352	2278	2398	2222	2330
Linien-Kosten	13.6	16.6	15,4	25,2	20.2	25.2	17.4
VM-ZfW bei							
späten Ereignissen	803 066	398 227	481 150	457 008	270 599	619 780	766 367
späten Aktivitäten	1 717 801	34 790	495 339	632 254	184 339	748 025	815 034

Man erkennt auf den ersten Blick keine Relation zwischen Zielfunktionswert des Linienverbundes nach unserem Maßstab und dem gemitteltem Zielfunktionswert aus dem Verspätungsmanagement. Es erscheint vollkommen unschlüssig, wieso H7 so eine gute Heuristik für das Verspätungsmanagement bei verspäteteten Aktivitäten sein soll, wo H6 eine so schlechte ist und selbst díe Optimallösung von Mosel kein besonders gutes Ergebnis abliefert.
Ebenso unschlüssig ist, wieso der zweite Split&Merge Verbund mit geringen Kosten und schlechterem Zielfunktionswert bessere Ergebnisse im Verspätungsmanagement erzielt als Cutting-Down oder der andere Split&Merge Verbund. Mit anderen Worten: Eine Stetigkeit in den Ergebnissen ist nicht abzulesen.

Die Datenbasis hier ist natürlich auch etwas spärlich. Vielleicht erbringen weitere numerische Untersuchungen mit größeren netzwerken andere Ergebnisse, die sich besser deuten lassen. Aber ein Ergebnis ist wohl dann doch deutlich: Es macht einen bisweilen sehr großen Unterschied für das Verspätungsmanagement, welchen Linienverbund bzw. welches Linienkonzept wir verwenden. Der Unterschied beläuft sich bei Faktor 2 bis Faktor 50!
Hier werden in den nächsten Monaten/Jahren vermutlich noch einige sehr interessante Ergebnisse zutage gebracht werden.

11 Linienkonzepte

Wie schon erwähnt worden ist, beschäftigte die Linienplanung sich meistens damit, aus einem Linienpool ein Linienkonzept zu erschaffen. Das bedeutet, man hat einfach jeder Linie des Pools eine Frequenz gegeben, wobei die Frequenz 0 dann bedeutete, dass die Linie im Verkehrsnetz nicht weiter auftauchte.
Wenn man aus unserem gewonnen Linienverbund ein Linienkonzept gewinnen wollen, brauchen wir im Grunde nur noch so einen Algorithmus auf unseren Linienverbund anzuwenden mit der zusätzlichen Einschränkung, dass alle Linien mindestens Frequenz 1 bekommen sollen. Wir haben uns ja zuvor stets allerbeste Mühe gegeben, dass die Linien im Verbund alle zusammen harmonieren. Würde irgendeine Linie ganz gestrichen, so würde unser Ergebnis im Grunde ganz zunichte gemacht werden.
Diese zusätzliche Bedingung, dass jede Frequenz im Linienkonzept mindestens 1 sein soll, ist auch weiter kein Problem für die bestehenden Probleme. Aber schauen wir uns einmal an, was da genau geschieht:

11.1 Bisherige Linienplanung

Der ganze Vorgang des Erstellens eines Linienkonzeptes nach [7] sieht jetzt so aus:

1. Erstelle einen Linienpool: bisher ist dies undefiniert geschehen, also wenig mathematisch.

2. Finde kürzeste Wege der Fahrgäste im PTN und trage diese additiv auf den Kanten ein: jeder Kante des PTN hat also nun eine ihr zugeordnete Zahl von Fahrgästen, die sie befahren.

3. Jede Kante des PTN bekommt eine upper- und lower-frequency in Abhängigkeit der gerade ermittelten Kunden-Kantengewichte zugeteilt: Alle Linien, die später das PTN befahren, müssen auf jeder Kante zusammen eine Frequenz haben, die zwischen upper- und lower-frequency liegt.

4. Finde jetzt Frequenzen zu den Linien, sodass die Kosten aller Linien minimiert werden, wobei die Frequenzen der Linien pro Kante jeweils zusammen zwischen upper- und lower-frequency liegen müssen: diese Aufgabe ist NP-schwer (Korollar 3.6 aus [7]).

Dabei sei zu bemerken, dass Punkt 4 NP-vollständig ist und Punkt 2 nur approximativ, weil die Fahrgäste sich endgültig für Umwege entschließen können, falls sie über die Linie, die den Umweg fährt, weniger umsteigen müssen. Das ist ja gerade der Ansatz des Change&Go Graphen.

Warum also wollen wir nicht den Change&Go Graphen dazu verwenden, die korrekten Wege im PTN der Fahrgäste zu berechnen? Der Gedanke liegt nahe, aber wenn eine Linie Frequenz 0 zugewiesen bekommen sollte, würden sich die Wege der Fahrgäste wieder ändern, weil diese Linie im Change&Go Graphen nicht auftauchen dürfte. Das bedeutet konkret, dass Wege und Linien mit Frequenz mindestens 1 miteinander korrellieren, weil nur diese Linien im C&G Graphen auftauchen dürfen.
Auf das eigentliche Linienkonzeptproblem lässt sich der C&G Graph also nicht direkt anwenden, was schade ist. Aber ursprünglich wollten wir ja sowieso fordern, dass in unserem Linienverbund jede Linie mindestens Frequenz 1 bekommen soll. Darauf wäre es dann doch anwendbar, weil dann die Wege im Change&Go Graphen immer noch alle existieren. Dies wollen wir jetzt einmal kurz vorstellen:

11.2 Vom Linienverbund zum Linienkonzept

Und jetzt können wir gleich den zweiten Schritt hinterher gehen: eigentlich interessieren uns bei dem Vorgang des Linienkonzeptes ja nicht die Fahrgäste auf den Kanten, sondern viel eher auf den Linien. Also was wir wollen, ist zu wissen, wieviele Fahrgäste zu einem Zeitpunkt wie dem Befahren einer Kante in einer Linie sitzen wollen. Dann müssen wir noch wissen, wieviele Personen ein Fahrzeug der Linie fassen kann und dann können wir mit einer simplen Modulo-Rechnung erfassen, welche Frequenz die Linie mindestens haben muss, um überhaupt alle Fahrgäste transportieren zu können. Hätte die Linie eine geringere Frequenz, würde es bei mindestens einer Kante des PTN zu eng im Fahrzeug werden. Eine höhere Frequenz ist zwar für eine Linie nicht schlecht, aber eben auch nicht nützlich, sondern nur teuer (die Frequenz einer Linie geht natürlich positiv in die Kostenfunktion ein). Wenn wir uns also die Mindestfrequenz als die endgültige Frequenz einer Linie definieren, haben wir auch zugleich eine Lösung des Linienkonzeptproblems mit minimalen Kosten.

11.2.1 Linienfrequenzen-Algorithmus

Input: PTN G, Linienverbund L und OD-Matrix W und Fahrgastgröße *vol* eines Fahrzeugs .

1. Erzeuge Change&Go Graph G' aus G und L.
2. Berechne die kürzesten Wege der OD-Paare.
3. Ermittle für jede Linie $l \in L$ die maximale Anzahl gleichzeitige Fahrgäste p_l.
4. Für alle $l \in L$ berechne $f_l := (p_l \mod vol) + 1$.

Output: L mit Frequenzvektor $(f_1, ..., f_{|L|})$.

Die einzige Frage, die sich noch stellt, ist: werden die upper- und lower-frequency der Kanten eingehalten? Diese Frage ist nicht außergewöhnlich wichtig. Erstens muss man erwähnen, dass diese upper- und lower-frequency meistens auch nur aus den Fahrgastdaten errechnet werden (bis auf verklehsberuhigte Zonen, die von der Infrastruktur gegeben werden). Aber zweitens stellen wir fest, dass ein Verstoß der Mindestfrequenzen gegen die upper-frequency nur bedeutet, dass das Problem ohnehin nicht lösbar ist. Und ein Verstoß gegen die lower-frequency lässt sich meistens leicht beheben, indem man die Frequenz einer oder mehrerer Linien leicht erhöht. Im Grunde sind die upper- und lower-frequencies also relativ unwichtig, wenn wir diese Methode anwenden, um die Frequenzen der Linien zu berechnen. Sie haben keine einschränkende Auswirkungen auf unser Ergebnis und entspringen sowieso nur einer Approximation-Rechnung, weil die kürzesten Wege nicht mit Change&Go Graph berechnet wurden.

12 Fazit

Die abschließende Frage ist: haben wir jene Lücke in der Verkehrsplanung geschlossen, die sich uns aufgetan hat? Haben wir es geschafft, nur aus PTN und OD-Matrix einen guten Linienverbund zu konstruieren? Die Antwort muss jein lauten.
Wir haben in den numerischen Ergebnissen gesehen, dass der Cutting-Down Algorithmus unser bestes Instrument ist, das Problem, einen optimierten Linienverbund zu konstruieren, hinreichend zu lösen. Aber für große Probleme ist Cutting-Down einfach nicht verwendbar, weil der Speicher schnell nicht ausreicht. Deswegen haben wir auch die meiste Zeit damit verbracht, den Split&Merge Algorithmus aufzubauen, denn der ist bei weitem nicht so speicherhungrig und kann auch auf gängigen Kleinrechnern die Probleme noch lösen. Zudem ist er auch deutlich schneller und kann große Probleme in einem für die Linienplanung verträglichen Zeitraum (Monaten) lösen.
Dieses Ergebnis ist schade, denn schon Cutting-Down ist eine Heuristik. Dass eine exakte Methode, das Problem zu lösen, zu lange brauchen würde, war uns stets klar. Aber dass unsere verwendbaren Heuristiken merkbar schlechter als Cutting-Down sind, die wiederum schlechter als das Optimum sein müsste, ist wie gesagt schade. Eine Lösung haben wir, aber sehr gut ist sie noch nicht.
Eine Hoffnung sollte uns sein, dass die Güte des Ergebnisses vom Split&Merge Algorithmus stark davon abhängt, welche Methode zur Erweiterung des Verbundes wir verwendet haben. Hier kann weitere Forschung mit Sicherheit noch deutlich bessere Ergebnisse erbringen.
Jetzt muss an sich noch eine gute Methode gefunden werden, um den Verbund in jedem Schritt zu erweitern. Die vorgeschlagene Greedy2-Methode arbeitet zwar ebenfalls recht akzeptabel, ist aber für große Probleme zu langsam. Es wäre schön, wenn wir dieses Problem ebenfalls mathematisch betrachten würden, um sowohl Laufzeit als auch Effizienz steigern zu können. Wir sind bisher an die Erkenntnis gelangt, dass wir dieses Problem als eine Art Rucksackproblem auffassen können, bei dem wir nicht wissen, welchen Wert die Gegenstände (also die Linien) haben, die wir in den Rucksack legen können. Das ist an sich ein interessantes Problem, das man sicherlich von vielen Perspektiven unterschiedlich beleuchten kann.
Was auf jeden Fall sinnvoll ist, ist die Anwendung des Change&Go Graphen. Wir haben mit seiner Hilfe es tatsächlich geschafft, einen Linienverbund mit sehr geringem Input zu finden. Auch das Berechnen des Linienkonzepts also der Frequenzen mit Change&Go Graphen ist deutlich einfacher und schöner als zuvor, was ja mehr eine Approximation gewesen ist.
Die große Laufzeit und der gigantische Speicherhunger der Algorithmen ist damit das größte Ärgernis. Allerdings muss man davon ausgehen, dass eine bessere Implementation als die hier verwendete viel bessere Geschwindgkeiten zulassen würde. Auch weil Linienplanung an sich nicht in zehn Minuten gelöst sein muss, sondern durchaus Monate Zeit hat, sind die Algorithmen und Methoden durchaus berechtigt.

Im Grunde haben wir uns wieder nur an jenes angeblich chinesische Sprichwort gehalten und ein großes Problem in mehrere kleinere Probleme zerteilt. Wir haben jedes kleine Problem für sich gelöst. Der Teufel steckt jetzt nur noch im Detail.

Übersicht über die Algorithmen

	Serielles Linienproblem	**Paralleles Linienproblem**
ohne Budget	H_0-Algorithmus H-Algorithmus	trivial Lösbar durch maximale Linienmenge
mit Budget		Split&Merge mit fester Kantenkardinalität Split&Merge mit Budget und • OD-Paar-Erweiterung • Greedy-Erweiterung • wirtschaftlicher Greedy-Erweiterung (Greedy2) • Basisnetz-Erweiterung (nicht implementiert) Cutting-Down

Eigenschaften der Algorithmen im Überblick:

- H_0: Verschlechtert Zielfunktionswert nie, optimale Lösung für Baumgraphen.
- H: Verschlechtert Zielfunktionswert nie, optimale Lösung für Baumgraphen, determiniert nach endlich vielen Schritten.
- Split&Merge mit fester Kantenkardinalität: Verschlechtert Zielfunktionswert nie, determiniert nach endlich vielen Schritten, nicht unbedingt optimal.
- Split&Merge mit Budget: Verschlechtert Zielfunktionswert nie, determiniert nach endlich vielen Schritten, nicht unbedingt optimal.
  - mit OD-Erweiterung: schnell, aber numerisch schlechte Ergebnisse.
  - mit Greedy-Erweiterung: Mäßig schnell, bessere numerische Ergebnisse als mit OD-Erweiterung, von Stochastik abhängig.
  - mit wirtschaftlicher Greedy-Erweiterung (Greedy2): Mäßig schnell, bessere numerische Ergebnisse als OD-Erweiterung und als Greedy, von Stochastik abhängig.
  - mit Basisnetz-Erweiterung: nicht implementiert, vermutlich aber schneller als Greedy-Erweiterung und mit ähnlicher Qualität.
- Cutting-Down: Verbessert Zielfunktionswert nie, arbeitet im unzulässigen Lösungsbereich (bis auf Endergebnis), keine Optimalitätsaussage, extrem lange Laufzeit, sehr viel Speicher nötig, aber gute numerische Ergebnisse (besser als Split&Merge im Mittel), von Stochastik abhängig.

Literatur

[1] Ahuja, Magnanti, Orlin: Network Flows; Theory, Algorithms, and Applications. 1993

[2] Florian Jarre & Josef Stoer: Optimierung, 2004

[3] L. Giovanni and G. Heilporn and M. Labbé: Optimization models for the delay management problem in public transportation. European Journal of Operational Research, 16.Sep.2008, Seiten 762-774.

[4] Hamacher, Horst & Klamroth, Kathrin: Lineare Optimierung und Netzwerkoptimierung. 2006, 2.Auflage

[5] C.B. Quak. Bus line planning. Master's thesis, TU Delft, 2003.

[6] Schöbel, Anita & Scholl, Susanne: Customer-Oriented Line Planning. dissertation.de, 2006

[7] Schöbel, Anita: Optimization Models in Public Transportation, 2004

[8] A.Schöbel: Integer Programming approaches for solving the delay management problem. Springer, 2007