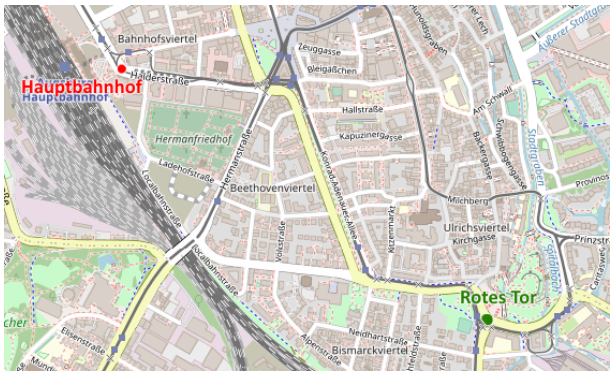
Universität Augsburg
Institut für Mathematik

Mai-Mathetag 2022

Kürzeste Wege berechnen

In diesem Brief wollen wir uns damit beschäftigen wie man kürzeste Wege berechnen kann – wir wollen uns also damit beschäftigen wie man (mathematisch) Fragestellungen der folgenden Form beantworten kann:

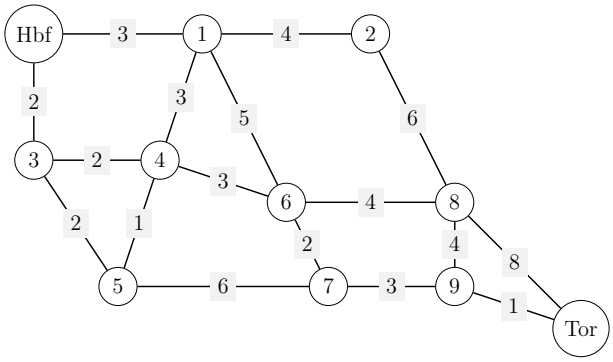
Wir sind am Augsburger Hauptbahnhof und möchten gerne zum Roten Tor. Was ist der kürzeste Weg dorthin?



Ein Ausschnitt aus dem Augsburger Stadtplan © OpenStreetMap-Mitwirkende

Wie würdest du laufen? Warum glaubst du, dass das der kürzeste Weg ist? Und wie hast du diesen Weg gefunden? Bist du dir sicher, dass es der kürzeste Weg ist?

Wie du vielleicht schon merkst, lassen sich diese Fragen so nicht wirklich mathematisch präzise beantworten. Wir müssen also erstmal noch das Problem etwas präziser formulieren. Als ersten Schritt wollen wir dazu den echten Stadtplan durch ein etwas übersichtlicheres Bild ersetzen:

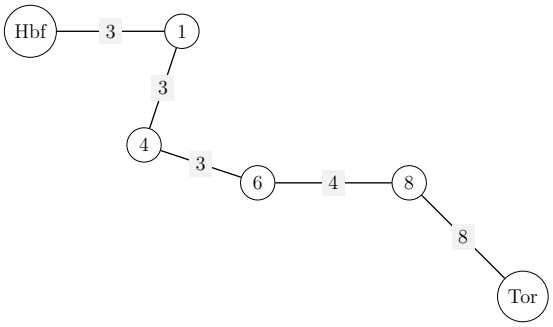


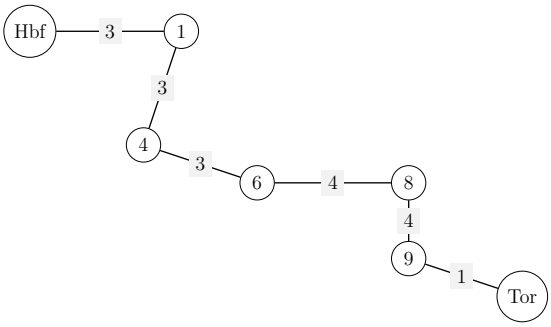
In diesem Bild haben wir die Straßen durch gerade Linien und die Kreuzungen durch Kreise dargestellt.[1] Außerdem haben wir auf jede Linie eine Zahl geschrieben – diese soll aussagen wie lang diese Straße ist, das heißt wie lange man von einer Kreuzung zur nächsten braucht.

So ein Bild nennt man in der Mathematik auch einen *Graphen*. Die Linien nennt man dann *Kanten* und die Kreise *Knoten*. Ein Weg ist jetzt einfach eine Abfolge von Kanten – beispielsweise ist Hbf $\to 1 \to 4 \to 6 \to 8 \to$ Tor ein möglicher Weg vom Hauptbahnhof zum Roten Tor (in unserem Graphen) und hat eine Länge von $3 + 3 + 3 + 4 + 8 = 21$. Unsere Aufgabe ist es jetzt einen *kürzesten* solchen Weg in dem Graphen zu finden.

Was meinst du? Ist das schon ein kürzester Weg? Oder findest du noch einen kürzeren Weg?

Tatsächlich ist unser aktueller Weg noch nicht optimal. Das können wir besonders einfach daran sehen, dass wir einen kürzeren Weg erhalten, indem wir eine Abkürzung gehen: Statt von 8 direkt zum Tor zu gehen, können wir über den Knoten 9 laufen – statt einer Länge von 8 haben wir dann nur noch eine Länge von $4 + 1 = 5$.





Auf diese Weise können wir jetzt immer weitermachen: Wir schauen, ob wir irgendwo eine Abkürzung finden und wenn ja, dann nehmen wir diese (und machen unseren Weg so kürzer). Das machen wir so lange, bis wir keine Abkürzung mehr finden. Und dann? Haben wir dann einen kürzesten Weg gefunden?

Dazu müssten wir uns ganz sicher sein, dass es keine weitere Abkürzung mehr gibt (dass wir also keine übersehen haben). Das klingt sehr schwierig das zu überprüfen. Insbesondere könnte eine Abkürzung ja auch ein völlig anderer (kürzerer) Weg sein! im Grunde müssen wir also *alle möglichen Wege* durchprobieren und schauen, welcher davon der kürzeste ist. Nur wenn wir das machen, können wir uns wirklich sicher sein, den kürzesten Weg gefunden zu haben. Aber ist das überhaupt machbar? Alle Wege durchprobieren?

Aufgabe 1.

Wie viele mögliche Wege gibt es in unserem Graphen?

a) Was schätzt du: Wie viele verschiedene Wege vom Hauptbahnhof zum Roten Tor gibt es in unserem Graphen?

[1] Damit das Bild nicht zu unübersichtlich wird habe ich zudem auch noch einige Straßen weggelassen.

b) Nehmen wir einmal kurz an, jeder der 12 Knoten wäre mit jedem anderen Knoten durch eine direkte Kante verbunden (das nennt man dann einen *vollständigen Graphen*). Wie viele Wege vom Hauptbahnhof zum Roten Tor gäbe es dann? Das ist eine obere Schranke an die Anzahl der Wege in unserem tatsächlichen Graphen. Warum?
Hinweis: Du darfst hier annehmen, dass ein Weg sinnvollerweise nie zweimal über die gleiche Kreuzung läuft. Bestimme jetzt wie viele Wege aus 1 Kante es gibt. Wie viele Wege aus 2 Kanten. Wie viele Wege aus 3 Kanten. Usw.

c) Nehmen wir nun stattdessen an die 12 Knoten wären in einem 3×4-Gitter angeordnet, wobei unser Startknoten ganz links oben und unser Zielknoten ganz rechts unten ist. Wie viele Wege gibt es jetzt, bei denen wir immer nur entweder nach rechts oder nach unten laufen? Das ist eine untere Schranke an die Gesamtzahl der Wege in unserem Gittergraphen. Warum?

d) Stell dir nun vor wir hätten einen Gittergraphen mit 10×10 Knoten. Was sind die untere und obere Schranke nun? Was ist mit einem $n \times n$ Gitter?

Offenbar ist es also nicht sinnvoll alle Wege durchzuprobieren! Wir brauchen daher ein sinnvolleres systematisches Verfahren: Einen *Algorithmus*! Ein Algorithmus ist dabei im Grunde nichts anderes als eine Zerlegung eines großen, komplexen Problems in viele kleine und einfach zu lösende Teilprobleme.

Im Folgenden werden wir zwei Algorithmen für das Kürzeste-Wege-Problem kennen lernen(es gibt natürlich noch viel mehr!)

1 Der Algorithmus von Bellman-Ford

Weil sich die Algorithmen so leichter erklären lassen, werden wir im Folgenden die Problemstellung leicht abwandeln: Wir bekommen weiterhin einen Graphen mit einem festen Startknoten (den wir ab sofort s nennen werden) gegeben. Statt aber nun einen konkreten kürzesten Weg zu einem festen Endknote zu bestimmen, wollen wir erstmal nur für jeden Knoten des Graphen seine *Distanz* vom Startknoten s bestimmen (also die Länge eines kürzesten Weges vom Startknoten zu diesem Knoten). Für einen Knoten v werden wir diese Distanz mit $d(v)$ bezeichnen (in unserem Augsburg-Graphen gilt also zum Beispiel $d(3) = 2$).

Der Algorithmus von Bellman und Ford basiert nun im Wesentlichen auf unserer Abkürzungsidee von vorhin: Wir starten mit „irgendwelchen“ Distanzen $\tilde{d}(v)$, für die wir uns sicher sind, dass wir die Knoten in dieser Zeit (oder schneller) erreichen können. Dann schauen wir immer wieder, ob es noch irgendwo eine Abkürzung gibt, durch die wir $\tilde{d}$ kleiner machen können. Das wollen wir dann so lange machen, bis wir uns sicher sein können, dass wir alle Abkürzungen gefunden haben.

Das ist jetzt aber natürlich immer noch etwas vage. Darum wollen wir die einzelnen Schritte jetzt genauer erklären:

- Als Startwerte können wir $\tilde{d}(s) = 0$ und $\tilde{d}(v) = \infty$ für alle anderen Knoten v wählen. Denn den Startknoten erreichen wir natürlich in 0 Zeit – und alle anderen sicher in unendlich Zeit (oder – hoffentlich – schneller).

- Statt alle nur möglichen Abkürzungen anzuschauen, betrachten wir nur Abkürzungen über einzelne Kanten. Wenn wir eine Kante $v-w$ (also eine Kante zwischen den Knoten v und w) haben, dann prüfen wir ob wir den Knoten w schneller erreichen können, indem wir zuerst zu v und dann von dort über die Kante $v-w$ zu w gehen. Falls dem so ist, passen wir $\tilde{d}(w)$ entsprechend an. Falls nicht, ändert sich nichts. Dann machen wir das gleiche mit vertauschten Rollen von v und w.

Ist dir klar, warum die obigen beiden Schritte korrekt sind? Warum gilt hier also zu jedem Zeitpunkt, dass es für jeden Knoten v einen Weg von s zu v gibt, dessen Länge höchstens $\tilde{d}(v)$ ist?

Wenn wir das verstanden haben, bleibt nur noch die Frage, wie oft wir diesen Abkürzungsschritt machen müssen, damit wir die tatsächlichen Distanzen bestimmt haben (also damit $\tilde{d}(v) = d(v)$ gilt)? Die Antwort auf diese Frage liefert der folgende Satz:

Satz 1. *Sei $s-v_1-v_2-v_3-\ldots-v_k$ ein kürzester Weg von s nach v_k. Wenn wir (in dieser Reihenfolge, aber möglicherweise noch mit anderen Schritten dazwischen) die Kanten $s-v_1, v_1-v_2, v_2-v_3, \ldots, v_{k-1}-v_k$ betrachten (und gegebenenfalls abkürzen), dann gilt danach $\tilde{d}(v_k) = d(v_k)$.*

Aufgabe 2.

Zeige, dass dieser Satz richtig ist.

Tipp: Zeige zunächst die folgende (einfachere) Aussage: Gilt $\tilde{d}(v) = d(v)$ und wir machen einen Abkürzungsschritt zur Kante $v-w$, dann gilt danach auch $\tilde{d}(w) = d(w)$. Überlege dir dann wie daraus der obige Satz folgt.

Mit diesem Satz können wir uns jetzt leicht überlegen wie oft wir Abkürzungsschritte ausführen müssen: Nehmen wir an unser Graph hat n Knoten. Dann besteht ein kürzester Weg sicher nicht aus mehr als $n-1$ Kanten (warum?). Zusammen mit dem obigen Satz folgt daraus, dass wir fertig sind, wenn wir $(n-1)$-mal jede Kante für einen Abkürzungsschritt betrachtet haben. Kannst du erkennen, warum das so ist?

Der vollständige Algorithmus lautet nun wie folgt:

Aufgabe 3.

Führe den Algorithmus von Bellman-Ford auf dem Beispiel-Graphen vom Anfang aus.

Aufgabe 4.

Wie oft musst du für unseren Beispiel-Graphen einen Abkürzungsschritt durchführen?

Wie oft müsstest du das bei einem Graphen aus n Knoten (höchstens) machen? Vergleiche dein Ergebnis mit dem aus Aufgabe 1.

Input : Ein Graph mit Längenangaben auf den Kanten und ein Startknoten s
Output : Für jeden Knoten v dessen Distant $d(v)$ von s aus
Setze $\tilde{d}(s) = 0$ und $\tilde{d}(v) = \infty$ für alle knoten $v \neq s$
for $i = 1, 2, \ldots, n-1$ **do**
  foreach *Kante* $v - w$ **do**
    if $\tilde{d}(v) +$ *Länge von* $v-w < \tilde{d}(w)$ **then**
      Setze $\tilde{d}(w) = \tilde{d}(v) +$ Länge von $v-w$
    end
    if $\tilde{d}(w) +$ *Länge von* $w-v < \tilde{d}(v)$ **then**
      Setze $\tilde{d}(v) = \tilde{d}(w) +$ Länge von $w-v$
    end
  end
end

2 Der Algorithmus von Dijkstra

Wir wir gesehen haben, ist der Algorithmus von Bellman-Ford deutlich schneller als einfach alle möglichen Wege durchzuprobieren. Allerdings ist er oft immer noch ziemlich ineffizient: So kommt es oft vor, dass wir die gleiche Kante immer wieder anschauen, ohne dadurch noch eine Abkürzung zu finden. Hier wäre es sehr hilfreich, wenn wir wüssten, wann wir mit einer bestimmten Kante fertig sind (diese also im weiteren Verlauf nicht mehr betrachten müssen). Der folgende Satz gibt uns einen sehr nützlichen Trick dafür:

Satz 2. *Sei v ein Knoten, für den $\tilde{d}(v) = d(v)$ gilt (wir kennen also bereits seine wahre Distanz vom Startknoten). Führen wir noch einen Abkürzungsschritt mit der Kante $v-w$ durch, so werden wir danach nie wieder eine Abkürzung über diese Kante finden.*

Aufgabe 5.

Zeige, dass der obige Satz wirklich wahr ist.

Jetzt wissen wir also, welche Kanten wir im Laufe des Algorithmus nicht mehr weiter betrachten müssen. Wir müssen nur noch herausfinden, für welche Knoten bereits $\tilde{d}(v) = d(v)$ gilt. Fällt dir ein Knoten ein, für den das bereits am Anfang des Algorithmus gilt?

Für unseren verbesserten Algorithmus werden wir die Knoten in zwei Gruppen einteilen – die fertigen Knoten und die unfertigen. Die fertigen Knoten sollen dabei die Eigenschaft haben, dass für sie bereits $\tilde{d}(v) = d(v)$ gilt und wir für jede mit ihnen verbundene Kante bereits den letzten Abkürzungsschritt gemacht haben. Wegen dem obigen Satz wissen wir dann, dass wir alle Kanten, die von einem fertigen Knoten ausgehen, im Weiteren ignorieren können. Diese Überlegung führt uns zum folgenden Algorithmus von Dijkstra:

Aufgabe 6.

Überzeuge dich, dass der obige Algorithmus wirklich korrekt ist.

Input : Ein Graph mit Längenangaben auf den Kanten und ein Startknoten s
Output : Für jeden Knoten v dessen Distant $d(v)$ von s aus
Setze $\tilde{d}(s) = 0$ und $\tilde{d}(v) = \infty$ für alle knoten $v \neq s$
Nenne alle Knoten „unfertig“
while *es gibt noch unfertige Knoten* **do**
  Wähle einen unfertigen Knoten v mit geringstem $\tilde{d}(v)$ unter allen unfertigen Knoten.
  foreach *Kante* $v - w$*, die von* v *ausgeht* **do**
    if $\tilde{d}(v) +$ *Länge von* $v{-}w < \tilde{d}(w)$ **then**
      Setze $\tilde{d}(w) = \tilde{d}(v) +$ Länge von $v{-}w$
    end
    if $\tilde{d}(w) +$ *Länge von* $w{-}v < \tilde{d}(v)$ **then**
      Setze $\tilde{d}(v) = \tilde{d}(w) +$ Länge von $w{-}v$
    end
  end
  Nenne den Knoten v „fertig“
end

Hinweis: Einen vollständigen Beweis hierfür zu finden, ist nicht ganz einfach. Aber vielleicht bist du ja schon davon überzeugt, dass der Algorithmus korrekt ist, wenn du die folgenden Fragen beantworten kannst:

- Warum kann sich der Wert $\tilde{d}(v)$ eines Knotens v nicht mehr ändern, nachdem wir ihn als fertig bezeichnet haben.
- Warum gilt für alle fertigen Knoten v, dass wir bereits ihre wahre Distanz von s kennen also $(\tilde{d}(v) = d(v))$?
- Ist ein Knoten v wirklich fertig, wenn wir ihn so nennen? Anders gefragt: Warum hat ein solcher Knoten die Eigenschaften, die in Satz 2 gefordert werden.

Aufgabe 7.

Führe den Algorithmus von Dijkstra auf dem Beispiel-Graphen vom Anfang aus.

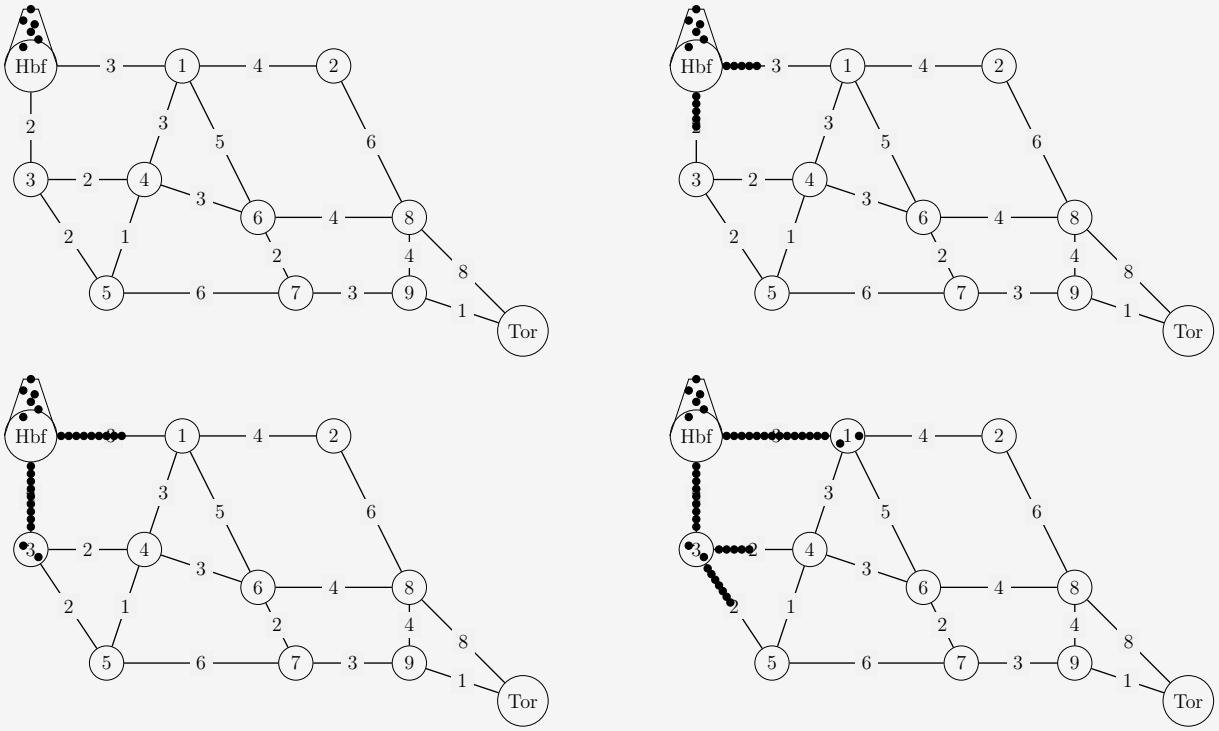
Aufgabe 8.

Wie oft musst du für unseren Beispiel-Graphen einen Abkürzungsschritt durchführen?

Wie oft müsstest du das bei einem Graphen aus n Knoten (höchstens) machen? Vergleiche dein Ergebnis mit dem aus Aufgabe 1 und Aufgabe 4.

Aufgabe 9.

Ich habe mal gelesen, dass der Dijkstra-Algorithmus auch als „Ameisen-Erkundungs-Algorithmus“ bezeichnet wird. Dieser Name kommt daher, dass man sich den Ablauf des Algorithmus auch so vorstellen kann:
Am Startknoten befindet sich ein großer Ameisenhügel, von dem aus sich die Ameisen zum Zeitpunkt 0 auf Erkundungstour machen und dazu gleichzeitig in alle Richtungen ausschwärmen. Die Ameisen bewegen sich dabei mit einer gleichmäßigen Geschwindigkeit (für eine Kante der Länge 6 brauchen sie also beispielsweise doppelt so lang wie für eine Kante der Länge 3 um vom einen Ende zum anderen zu kommen). Wann immer die Ameisen zu einem Knoten kommen, an dem noch keine anderen Ameisen sind, teilen sie sich auf und strömen in alle ausgehenden Kanten um weiter zu erkunden. Kommen sie an einen Knoten, an dem bereits andere Ameisen sind, hören sie auf in diese Richtung zu suchen (weil das ja schon die Ameisen übernommen haben, die diesen Knoten früher erreicht haben). Im Folgenden habe ich mal ein paar Beispiele wie das ganze nach Zeit 0, 1, 2 und 3 in unserem Beispiel-Graphen aussieht (die Punkte sollen die Ameisen sein):



Wenn wir jetzt die Ameisen von oben beobachten und uns die Zeiten aufschreiben, an denen die Ameisen zum ersten mal einen Knoten erreichen, dann erhalten wir genau unsere Distanzen vom Startknoten. Kannst du erkennen, warum das so ist? Lasse die Ameisen weiterlaufen und versuche so die Distanzen zu bestimmen.

Siehst du den Zusammenhang zum Algorithmus von Dijkstra?

3 Weitere Aufgaben

Aufgabe 10.

Eigentlich wollten wir doch kürzeste Wege bestimmen – unsere beiden Algorithmen berechnen aber nur kürzeste *Distanzen* (die Werte $d(v)$)! Hilft uns das überhaupt weiter?

Tatsächlich ist es nicht besonders schwierig kürzeste Wege zu bestimmen, wenn man erstmal die kürzesten Distanzen kennt. Kannst du herausfinden wie?

Hinweis: Es gibt (mindestens) zwei Wege das zu tun:

- Entweder du versuchst die Algorithmen so anzupassen, dass wir uns dabei schon irgendwie *merken* wie wir jeweils zu den einzelnen Knoten gekommen sind.
- Oder du nutzt die Distanzen, die wir am Ende bestimmt haben um daraus *rückwärts* kürzeste Wege zu den einzelnen Knoten zu konstruieren.

Aufgabe 11.

Denke dir selbst einen Graphen aus und bestimme darin kürzeste Wege.

Aufgabe 12.

Finde Graphen, auf denen unsere Algorithmen möglichst schlecht sind. Also zum Beispiel einen Graphen, auf dem der Wert $\tilde{d}(v)$ ein und desselben Knotens v möglichst oft geändert wird. Oder ein Graph mit einer Kante $v-w$, die möglichst oft zu einer neuen Abkürzung führt.

Was ist die höchste Zahl die du erreichen kannst?

Aufgabe 13.

Bisher waren die Längen der Kanten immer positive Zahlen – was aber passiert, wenn wir auch negative Zahlen zulassen? [2] Funktionieren unsere Algorithmen dann immer noch, oder kannst du dir einen Graphen überlegen, in dem die Algorithmen dann etwas falsches machen?

Hinweis: Das ist eine sehr offene Frage, die nicht eine eindeutige richtige Antwort hat, aber zu vielen interessanten neuen Fragen führen kann :-)