Algorithmen auf Graphen

Hans-Jörg Kreowski

Universität Bremen
Studiengang Informatik
kreo@informatik.uni-bremen.de

Wintersemester 2003/04

Inhaltsverzeichnis

Kapitel 1

Graphen in der Informatik

Wo in der Informatik schwierige Zusammenhänge veranschaulicht oder kompliziert strukturierte Datenobjekte bearbeitet und untersucht werden sollen, sind Graphen als Darstellungsmittel beliebt. *Graphen* sind Strukturen, mit denen sich Beziehungen zwischen Einzelheiten ausdrücken lassen. Es gibt Graphen in vielen Variationen; denn ihre drei Grundbestandteile – Knoten, Kanten und Markierungen – können unterschiedlich miteinander verknüpft werden.

In allen Graphenbegriffen kommen Knoten und Kanten vor. *Knoten* sind voneinander unterscheidbare, eigenständige Objekte; Kanten verbinden Knoten. Während Knoten als Elemente von (Knoten-)Mengen in praktisch allen Graphenbegriffen geeignet identifizierbare Objekte sind, können Kanten sehr verschiedenartig gewählt werden, um Beziehungen zwischen Knoten auszudrücken:

(a) Kann eine Kante Beziehungen zwischen beliebig vielen Knoten herstellen, spricht man von einer *Hyperkante*. Dementsprechend werden Graphen mit Hyperkanten *Hypergraphen* genannt. Üblicher jedoch ist, dass eine Kante zwei Knoten verbindet, was im folgenden behandelt wird.

(b) Sind die beiden Knoten einer Kante gleichberechtigt, handelt es sich um eine *ungerichtete Kante*. Andernfalls ist die Kante *gerichtet*. Technisch lässt sich eine Kante dadurch „richten“, dass ihr ein geordnetes Paar von Knoten zugeordnet wird oder die beiden Knoten verschiedene Bezeichnungen erhalten. Üblich ist die Unterscheidung durch Angabe von *Quelle* und *Ziel* einer Kante. Ein Graph, der nur aus ungerichteten Kanten besteht, wird selbst *ungerichtet* genannt. Analog enthält ein *gerichteter Graph* ausschließlich gerichtete Kanten.

(c) Außerdem können Kanten wie Knoten individuelle Objekte sein; die zugehörigen Knoten müssen dann gesondert zugeordnet werden. Oder Kanten sind aus Knoten zusammengesetzte Größen. Im ersten Fall kann es in einem Graphen gleichartige parallele Kanten zwischen zwei Knoten geben, im zweiten Fall nicht.

Markierungen schließlich dienen dazu, Knoten und Kanten mit zusätzlicher Information zu versehen. Es gibt vier Standardmöglichkeiten, mit Markierungen umzugehen. Sie völlig zu verbieten, liefert *unmarkierte Graphen*. Nur die Markierung von Knoten zu erlauben, ergibt

Knoten–markierte Graphen. Analog werden bei *Kanten–markierten Graphen* nur Kanten markiert. Werden Knoten und Kanten markiert, erhält man *markierte Graphen.*

Diese verwirrende Vielfalt hat den Vorteil, dass das Konzept der Graphen flexibel an konkrete Anwendungssituationen angepasst werden kann. Aber die Variabilität hat auch ihren Preis. Da die verschiedenen Graphenbegriffe unterschiedliche mathematische Eigenschaften aufweisen, lassen sich Ergebnisse über die eine Art von Graphen nicht ohne weiteres auf eine andere Art übertragen:

WARNUNG: Graph ist nicht gleich Graph!

Einige Beispiele sollen die Bandbreite von Graphen und ihre Anwendungsmöglichkeiten illustrieren.

1.1 Wohlstrukturierte Flussdiagramme

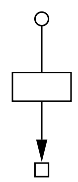
Wohlstrukturierte Flussdiagramme sind graphische Darstellungen der Anweisungen einer einfachen, PASCAL–ähnlichen Programmiersprache. Die Syntax sieht Wertzuweisung, Reihung, Fallunterscheidung (*if–then–else*) und Wiederholung (*while–do*) vor; die zugehörige Backus–Naur–Form lautet:

$$S \ ::= \ V := E \mid S;S \mid \mathit{if}\ B\ \mathit{then}\ S\ \mathit{else}\ S \mid \mathit{while}\ B\ \mathit{do}\ S$$

Dabei steht S für „statement“, V für „variable“, E für „expression“ und B für „boolean expression“. Die entsprechenden Flussdiagramme können folgendermaßen als Graphen beschrieben werden:

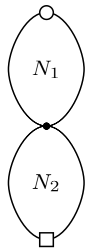
(a) Sie haben immer genau einen Eingang und einen Ausgang, die Knoten sind und mit *begin* bzw. *end* bezeichnet werden.

(b) Eine Wertzuweisung wird als Kante dargestellt:

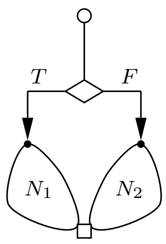


Der Kasten in der Mitte der Kante kann als Markierung aufgefasst werden. Er dient lediglich der Verstärkung des optischen Eindrucks, damit die beschriebenen Graphen auch wie Flussdiagramme aussehen und diese nicht nur formal darstellen. Man beachte, dass nur die Tatsache repräsentiert ist, dass eine Wertzuweisung stattfindet, nicht jedoch, welche Variable welchen Wert erhält. (Das ließe sich durch eine entsprechende Markierung der Kante erreichen.)

(c) Ein Graph N ist ein Flussdiagramm, das eine Reihung repräsentiert, wenn er sich in zwei Graphen N_1 und N_2 aufteilen lässt, die beide Flussdiagramme darstellen, derart dass der Eingang von N_1 der Eingang von N, der Ausgang von N_2 der Ausgang von N und der Ausgang von N_1 und der Eingang von N_2 der einzige gemeinsame Knoten von N_1 und N_2 in N ist:

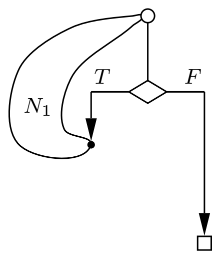


(d) Ein Graph N ist ebenfalls ein Flussdiagramm, das der Fallunterscheidung entspricht, wenn er sich folgendermaßen aufteilen lässt:



Dabei müssen N_1 und N_2 Flussdiagramme sein, deren Ausgang auch Ausgang von N ist und die sonst keine gemeinsamen Knoten haben. Außerdem muss 1 der Eingang von N_1 sein und 2 der von N_2. Neben seinem Eingang muss N noch einen weiteren Knoten besitzen, der mit der Rhombusform markiert ist. Vom Eingang führt eine Kante in diesen Knoten. Von diesem Knoten aus geht je eine Kante nach 1 und 2. Eine davon ist mit T (für „true") markiert, die andere mit F (für „false").

(e) Schließlich ist ein Graph N ein Flussdiagramm, das für die Wiederholung steht, wenn er folgende Form hat:



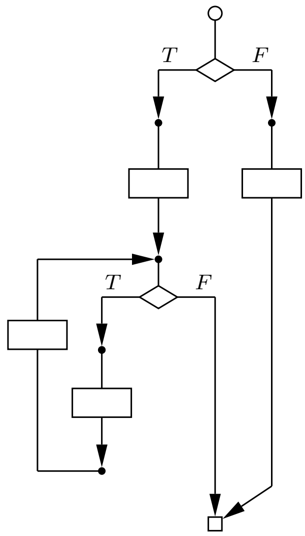
Dabei muss N_1 ein Flussdiagramm sein, dessen Eingang der Knoten 0 und dessen Ausgang der Eingang von N ist. Außerdem besitzt N neben seinem Ausgang noch einen Knoten mit drei anhängenden Kanten wie bei den Fallunterscheidung, wobei hier allerdings die mit F markierte Kante zum Ausgang geht und die mit T markierte zum Knoten 0.

Beachte, dass in (d) und (e) von den konkreten Booleschen Ausdrücken abstrahiert wird.

Ein Programm, das die n-te Potenz von x folgendermaßen bestimmt:

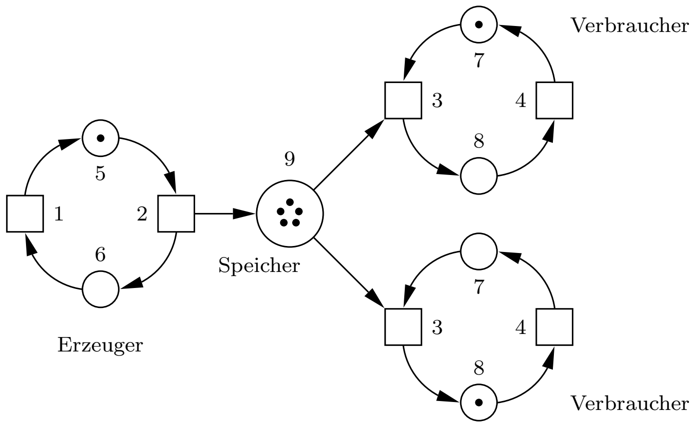
```
if    n > 0
then  exp := 1;
      while n > 0 do n := n - 1;
                     exp := exp * x
else  exp := 0
```

besitzt als Flussdiagramm den Graphen:



1.2 Erzeuger-Verbraucher-System

Situationen, die in technischen oder informationsverarbeitenden Systemen auftreten, lassen sich sehr häufig angemessen durch Petri-Netze darstellen, die spezielle Graphen sind. Das soll anhand eines vereinfachten Warenlagers demonstriert werden, das von einem Erzeuger bestückt wird und aus dem zwei Verbraucher beliefert werden. Der folgende Graph hält einen „Schnappschuss" aus diesem System fest:



Dabei gehören die Ziffern nicht zum eigentlichen Graphen, sondern dienen als Referenzen, um über den Sinn der einzelnen Knoten besser sprechen zu können. Alle eckig eingefassten Knoten sind gleich markiert, dazu kann das graphische Objekt einfach als Markierung aufgefasst werden. Diese Knoten repräsentieren jeweils mögliche Aktivitäten des Systems; die folgende Tabelle weist aus, wofür jeder Knoten, abhängig von seiner Ziffer, steht:

1. Vorbereitung (eines Produktionsvorgangs)
2. Herstellung (einer Wareneinheit)
3. Erwerb (einer Wareneinheit)
4. Erwerbsvorbereitung

Die als Kreis gezeichneten Knoten stellen Systemeigenschaften oder -zustände dar. Ihre Markierung ist immer eine natürliche Zahl, die Zahl der im Kreis befindlichen Punkte. Ihr Wert sagt etwas über die konkrete Beschaffenheit der jeweiligen Eigenschaften. Im einzelnen beschreiben die Knoten, die mit 5 bis 9 nummeriert sind, folgende Aspekte des Systems:

5. produktionsbereit
6. nicht produktionsbereit
7. empfangsbereit
8. nicht empfangsbereit
9. Warenlager

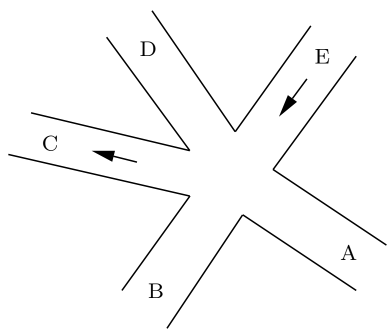
Für die Knoten 5 bis 8 bedeutet die Markierung 0, dass die genannte Eigenschaft gerade nicht vorliegt, während 1 die Erfülltheit beschreibt. Alle anderen Zahlen machen keinen Sinn. Die Markierung des Knotens 9 gibt die Zahl der momentan im Warenlager gespeicherten Wareneinheiten an.

Es ist klar, dass analog zum obigen Graphen durch Veränderung der natürlichzahligen Markierungen andere Systemsituationen repräsentiert werden.

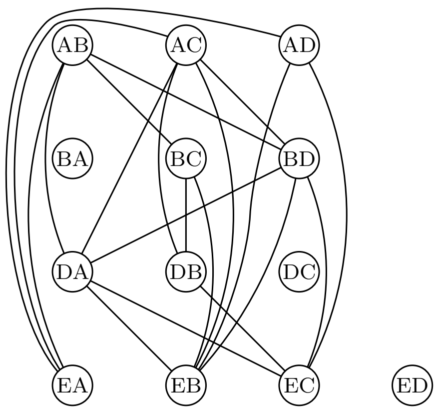
1.3 Verkehrsfluss auf Kreuzungen

Graphen lassen sich auch vorteilhaft einsetzen, wenn bestimmt werden soll, wie viele Ampelphasen erforderlich sind, um komplizierte Kreuzungen verkehrsgerecht zu steuern.

Betrachte etwa folgende Kreuzungssituation:



Die Zufahrtsstraßen sind mit A bis E bezeichnet, und die Fahrtrichtung von Einbahnstraßen ist durch einen zusätzlichen Pfeil angegeben. Paare von Buchstaben beschreiben dann Verkehrsströme, die die Kreuzung passieren können. Sie werden als Knoten eines Graphen genommen. Einige Verkehrsströme können nicht gleichzeitig über die Kreuzung fahren, weil sie sich kreuzen. Wenn das für zwei Verkehrsströme gilt, wird eine Kante (beliebiger Richtung) zwischen ihren Knoten gezogen. Für die obige Kreuzung entsteht so der Graph:



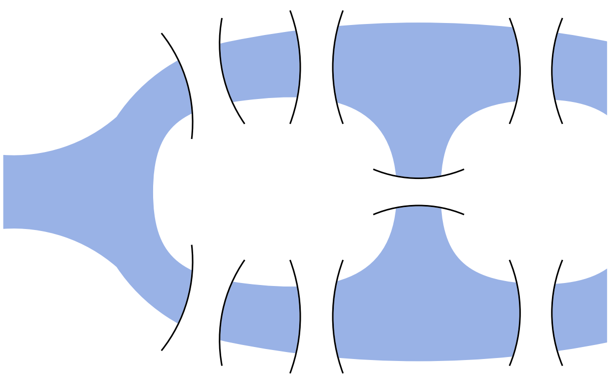
Solche Graphen enthalten wichtige Informationen für die Planung der Ampelphasen an den zugrundeliegenden Kreuzungen, weil grünes Licht für einen Verkehrsstrom als Konsequenz hat, dass alle Verkehrsströme, die mit dem einen durch eine Kante verbunden sind, vor roten Ampeln halten müssen.

Kapitel 2

Eulersche Graphen

Eine der Wurzeln der mathematischen Graphentheorie ist eine 1736 erschienene Arbeit von Euler über das Königsberger Brückenproblem. Die Lösung stellt gleichzeitig ein interessantes und typisches Beispiel für einen Algorithmus auf Graphen dar.

Dem Problem liegt ein Ausschnitt aus dem Königsberger Stadtplan zugrunde. Dort wo der alte und der neue Arm des Pregel zusammenfließen, bilden sie zusätzlich eine Insel, so dass der Fluss vier Landbereiche voneinander trennt, die durch insgesamt sieben Brücken verbunden waren, wie die folgende Skizze zeigt.



Euler hat sich gefragt, ob es einen Rundweg gibt, bei dem jede der sieben Brücken genau einmal überquert wird. Seine Antwort ist NEIN, und er hat sie nicht nur für die Königsberger Situation gegeben, sondern ein Verfahren entwickelt, mit dem sich die Frage für jeden beliebigen Wegeplan beantworten lässt.

Dazu hat er die Landbereiche als Knoten aufgefasst und die Brücken als ungerichtete Kanten zwischen den Landbereichsknoten, die sie verbinden. Bei dieser Abstraktion entsteht ein ungerichteter Graph, dessen Kanten an je zwei Knoten aufgehängt sind. Für solche Graphen lautet dann die Frage:

> Gibt es einen Rundweg (Kreis), der jede Kante genau einmal durchläuft?
>
> Wenn JA, wird der Graph Eulersch genannt.

2.1 Ungerichtete Graphen

Ein *ungerichteter Graph* ist ein System $G = (V, E, att\colon E \to \binom{V}{2})$ [1], wobei V eine endliche Menge von *Knoten*, E eine endliche Menge von *Kanten* und *att* eine Abbildung ist, die *Aufhängung* heißt und jeder Kante zwei Knoten zuordnet.

Da jede Kante an zwei verschiedenen Knoten hängt, gibt es nach dieser Definition keine Schleifen, also keine Kanten, die mit beiden Enden am selben Knoten hängen. Erlaubt aber sind parallele Mehrfachkanten, denn als Abbildung kann die Aufhängung verschiedenen Kanten dieselben Knoten zuordnen.

Im Zusammenhang mit der Aufhängung gibt es noch zwei Sprechweisen, die häufig benutzt werden. Ein Knoten v ist *inzident* mit einer Kante e, falls $v \in att(e)$, und zwei Knoten v und v' sind *adjazent*, falls $att(e) = \{v, v'\}$ für eine Kante e gilt.

Für Beispiele wird meist statt der mengentheoretischen Repräsentation eines Graphen gemäß Definition eine graphische gewählt, bei der die Knoten als dicke Punkte, Kreise, Rechtecke oder ähnlich dargestellt werden und die Kanten als Linien gezeichnet sind, deren Endpunkte gerade die Aufhängungsknoten verbinden.

2.2 Wege und Eulersche Kreise

Sei $G = (V, E, att)$ ein ungerichteter Graph. Dann ist ein *Weg von v nach v'* eine alternierende Knoten-Kanten-Folge $v_0 e_1 v_1 e_2 \cdots e_n v_n$ mit $v = v_0$, $v' = v_n$ und $att(e_i) = \{v_{i-1}, v_i\}$ für $i = 1, \ldots, n$. Ein *Kreis* (durch v) ist ein Weg von v nach v mit $e_i \neq e_{i+1}$ für $i = 1, \ldots, n-1$ und $e_1 \neq e_n$. Ein *Eulerscher Kreis* ist ein Kreis mit $e_i \neq e_j$ für $i \neq j$ und $\{e_1, \ldots, e_n\} = E$. G ist *Eulersch*, falls ein Eulerscher Kreis in G existiert. G ist *zusammenhängend*, falls ein Weg von v nach v' für je zwei Knoten $v, v' \in V$ existiert.

2.3 Einfache Eulersch-Tests – zu aufwändig

Da ein Eulerscher Kreis alle Kanten je genau einmal durchläuft, ist er eine Permutation der Kantenmenge und ein Weg der Länge n, wenn das die Kantenzahl ist. Einfache Möglichkeiten, für einen Graphen zu testen, ob er Eulersch ist oder nicht, bestehen somit in folgenden Verfahren:

(1) Probiere alle Permutationen der Kantenmenge oder

(2) probiere alle Wege der Länge n mit $n = \#E$ [2], ob sie einen Eulerschen Kreis bilden.

---

[1] $\binom{V}{2}$, was als „V über 2" gelesen wird, bezeichnet die Menge aller 2-elementigen Teilmengen von V. Die Schreibweise ist an die bekannte Notation $\binom{n}{2}$ angelehnt, denn $\binom{n}{2}$ ist die Zahl der Elemente von $\binom{V}{2}$, wenn V n Elemente hat.

[2] Für eine endliche Menge X bezeichnet $\#X$ die Zahl ihrer Elemente.

Das Problem dieser und ähnlicher Verfahren ist, dass es exponentiell viele Permutationen und zumindest im schlechtesten Fall auch exponentiell viele Wege gibt. Genaue Überlegungen zur Zahl der Wege in einem Graphen werden später angestellt. Verfahren dieser Art werden *ausschöpfende Suche* genannt und erstaunlich häufig in Expertensystemen u.ä. verwendet. Mit dieser Art der Suche gibt es allerdings in aller Regel Effizienzprobleme, weil die Suchräume meist zumindest exponentiell groß sind.

Es lohnt sich also, nach einem Eulersch-Test zu fahnden, der schnell geht. Es ist nicht schwer zu sehen, dass Eulersche Graphen gradgradig und bis auf isolierte Knoten zusammenhängend sind. Dabei bedeutet gradgradig, dass jeder Knoten mit einer geraden Zahl von Kanten inzident ist, was dadurch zustandekommt, dass ein Eulerscher Kreis bei jedem Besuch eines Knotens mit einer Kante hineingeht und mit einer anderen wieder hinaus.

2.4 Beobachtung zur Gradgradigkeit

Ein Eulerscher Graph ist gradgradig.

Ein Graph G ist gradgradig, falls $\mathit{degree}(v) \mod 2 = 0$ für alle $v \in V$, wobei der *Grad* eines Knoten v definiert ist durch $\mathit{degree}(v) = \#\{e \in E \mid v \in \mathit{att}(e)\}$.

Man beachte, dass die Knotengrade leicht zu berechnen sind, indem man für alle Kanten die Aufhängung anschaut und für die jeweils beiden Knoten einen Zähler um 1 erhöht, der anfangs auf 0 steht.

Beweis(skizze)

Sei $v_0 e_1 \cdots e_n v_n$ ein Eulerscher Kreis und $v \in V$ mit $\mathit{degree}(v) > 0$ ein Knoten, der k-mal in den Schritten $i_1 < i_2 < \cdots < i_k$ besucht wird. Dann sind die Kanten $e_{i_1}, e_{i_1+1}, e_{i_2}, e_{i_2+1}, \ldots, e_{i_k}, e_{i_k+1}$ mit v inzident. Da es sich um einen Eulerschen Kreis handelt, sind je zwei dieser Kanten verschieden und alle mit v inzidenten Kanten kommen vor, so dass $\mathit{degree}(v) = 2k$. Im Falle von $i_k = n$ ist die letzte Kante e_1. Ansonsten gilt $\mathit{degree}(v) = 0$, so dass insgesamt alle Knoten geraden Grad besitzen.

2.5 Beobachtung zum Zusammenhang

Sei $G = (V, E, \mathit{att})$ ein Eulerscher Graph und G^0 der größte Teilgraph von G ohne isolierte Knoten, d.h. $G^0 = (V - V_0, E, \mathit{att}_0)$ mit $V_0 = \{v \in V \mid \mathit{degree}(v) = 0\}$ und $\mathit{att}_0(e) = \mathit{att}(e)$ für alle $e \in E$. Dann ist G^0 zusammenhängend.

Beweis(skizze)

Je zwei Knoten auf einem Kreis sind durch mindestens 2 Wege (Kreishälften) miteinander verbunden. Auf einem Eulerschen Kreis liegen aber alle Knoten mit positivem Grad, also alle Knoten von G^0, so dass sich G^0 als zusammenhängend erweist.

2.6 Eulersch-Test

Die Lösung des Königsberger Brückenproblems für beliebige ungerichtete Graphen besteht darin, dass die beiden beobachteten Eigenschaften Eulerscher Graphen nicht nur notwendig sind, sondern auch hinreichend. Das ergibt folgenden schnellen Algorithmus:

Eulerian-test

Eingabe: $G = (V, E, att)$ mit $E \neq \emptyset$.

Verfahren: (1) Bestimme Grad aller Knoten,
(2) bilde G^0,
(3) prüfe Zusammenhang von G^0.

Ausgabe: JA, falls G gradgradig und G^0 zusammenhängend ist,
NEIN sonst.

Der Aufwand dieses Algorithmus ist polynominell. Denn um den Grad aller Knoten zu bestimmen, muss man für jede Kante zwei Zähler hochstellen und am Ende für jeden Knoten die Gradheit des Grades prüfen. Um G^0 zu bilden, muss man die isolierten Knoten löschen, die ja durch die Gradbestimmung bereits bekannt sind. Und schließlich muss der Zusammenhang ermittelt werden, was bezogen auf die Knotenzahl kubisch viel Zeit beansprucht (vgl. Kapitel Kürzeste Wege).

Der Eulersch-Test prüft, ob der Eingabegraph gradgradig und bis auf isolierte Knoten zusammenhängend ist. Es soll gezeigt werden, dass dieser Algorithmus korrekt ist. Nach den bisherigen Beobachtungen haben Eulersche Graphen die genannten Eigenschaften, so dass nur noch die Umkehrung nachgewiesen werden muss. Das soll in vier Schritten geschehen:

1. Ein gradgradiger Graph mit Kanten besitzt einen (einfachen) Kreis.
2. Löscht man die Kanten dieses Kreises, entsteht wieder ein gradgradiger Graph.
3. Damit lässt sich der Vorgang wiederholen, bis alle Kanten auf Kreise verteilt sind.
4. Aus der so gewonnenen Kantenüberdeckung kann ein Eulerscher Kreis konstruiert werden, wenn der betrachtete Graph bis auf isolierte Knoten zusammenhängt.

2.7 Konstruktion eines Kreises in gradgradigen Graphen

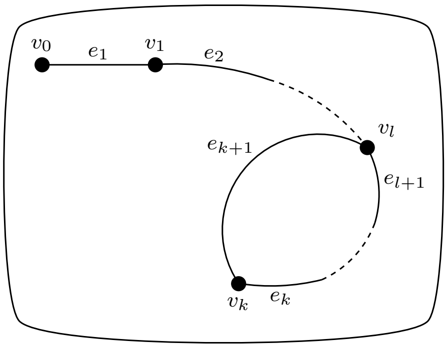
Betrachte einen einfachen Weg $v_0e_1 \cdots e_kv_k$ in G, wobei einfach heißt, dass je zwei Knoten auf dem Weg verschieden sind, also $v_i \neq v_j$ für $i \neq j$. Da e_k mit v_k inzident ist, hat v_k einen positiven Grad: $degree(v_k) > 0$. Wegen der Gradgradigkeit impliziert das $degree(v_k) \geq 2$ und somit die Existenz einer Kante e_{k+1} mit $e_{k+1} \neq e_k$ und $att(e_{k+1}) = \{v_k, v_{k+1}\}$. Für v_{k+1} gibt es zwei Fälle:

1. Fall: $v_{k+1} = v_l$ für ein $l \in \{0, \ldots, k-1\}$.

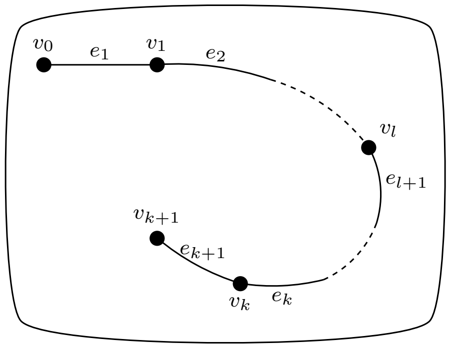
Dann ist $v_le_{l+1} \cdots e_{k+1}v_{k+1} = v_l$ ein einfacher Kreis, wobei ein Kreis einfach ist, wenn bei Löschen einer Kante ein einfacher Weg entsteht.

2. Fall: $v_{k+1} \neq v_l$ für alle $l = 0, \ldots, k$.

Dann ist $v_0 e_1 \cdots e_k v_k e_{k+1} v_{k+1}$ ein einfacher Weg, der den ursprünglichen um 1 verlängert.



1. Fall: $v_{k+1} = v_l$



2. Fall: $v_{k+1} \neq v_l$

In einem gradgradigen Graphen lässt sich also ein einfacher Weg entweder verlängern oder daraus ein einfacher Kreis gewinnen. Wenn der Graph Kanten hat, kann man mit dem einfachen Weg $v_0 e_1 v_1$ beginnen, wobei e_1 eine beliebige Kante ist. Dieser Weg lässt sich höchstens bis zur Länge $\#V - 1$ verlängern. Denn ein Weg $v_0 e_1 \cdots e_n v_n$ für $n = \#V$ besucht $\#V + 1$ Knoten, so dass mindestens ein Knoten doppelt vorkommt. Wenn sich der betrachtete Weg nicht mehr verlängern lässt, ergibt sich wie gewünscht ein einfacher Kreis.

2.8 Löschen eines Kreises in gradgradigen Graphen

Sei $k_0 = \bar{v}_0 \bar{e}_1 \cdots \bar{e}_m \bar{v}_m = \bar{v}_0$ ein einfacher Kreis in G. Dann ist $G - k_0 = (V, E - E(k_0), att|_{E-E(k_0)})$ mit $E(k_0) = \{\bar{e}_1, \ldots, \bar{e}_m\}$ ein Teilgraph[3] von G. Dabei ist die Aufhängung von Kanten die ursprüngliche Aufhängung eingeschränkt auf die kleinere Kantenmenge. Nach Konstruktion gilt für die Knotengrade in $G - k_0$ und G:

$$\begin{aligned} \mathit{degree}_{G-k_0}(\bar{v}_i) &= \mathit{degree}_G(\bar{v}_i) - 2 \text{ für } i = 1, \ldots, m \text{ und} \\ \mathit{degree}_{G-k_0}(v) &= \mathit{degree}_G(v) \text{ sonst.} \end{aligned}$$

Somit ist $G - k_0$ gradgradig, wenn es G ist.

2.9 Konstruktion einer Kantenüberdeckung für gradgradige Graphen

Der Vorgang des Konstruierens und Löschens einfacher Kreise lässt sich also wiederholen, bis keine Kanten mehr übrig sind. So entsteht eine Kantenüberdeckung. Eine Menge $\mathcal{K}$ von einfachen Kreisen in G ist eine *Kantenüberdeckung*, wenn jede Kante von G in genau einem Kreis von $\mathcal{K}$ vorkommt, d.h. $E(k_1) \cap E(k_2) = \emptyset$ für alle $k_1, k_2 \in \mathcal{K}$ mit $k_1 \neq k_2$ und $\bigcup\limits_{k \in \mathcal{K}} E(k) = E$.

[3] Ein Graph $G' = (V', E', att')$ ist Teilgraph des Graphen $G = (V, E, att)$, wenn $V' \subseteq V, E' \subseteq E$ und $att'(e) = att(e)$ für alle $e \in E'$.

Die obigen Überlegungen ergeben, dass jeder gradgradiger Graph eine Kantenüberdeckung besitzt, die auch algorithmisch konstruiert werden kann. Das gilt auch noch für Graphen ohne Kanten, die von der leeren Kantenüberdeckung überdeckt werden (wenn man $\bigcup_{k \in \emptyset} E(k) = \emptyset$ wählt).

2.10 Konstruktion Eulerscher Kreise aus Kantenüberdeckungen

Sei nun G nicht nur gradgradig, sondern auch bis auf isolierte Knoten zusammenhängend. Sei ferner $\mathcal{K}$ eine Kantenüberdeckung, die nicht leer ist (so dass auch E nicht leer ist). Mit vollständiger Induktion über die Zahl der Kreise in $\mathcal{K}$ soll bewiesen werden, dass ein Eulerscher Kreis konstruiert werden kann.

I.A.: $\mathcal{K} = \{k_0\}$. Dann ist k_0 Eulerscher Kreis.

I.V.: Die Behauptung gelte für $\mathcal{K}$ mit n Kreisen.

I.S.: Betrachte $\mathcal{K} = \{k_1, \dots, k_{n+1}\}$ mit $n \geq 1$.

Es sei zusätzlich vorausgesetzt, dass die Kreise so angeordnet sind, dass jeder mit zumindest einem seiner Vorgänger Knoten gemeinsam hat, d.h. für $m = 1, \dots, n$ ist

$$V(k_{m+1}) \cap \bigcup_{i=1}^{m} V(k_i) \neq \emptyset,$$

wobei $V(k)$ die Menge der in k vorkommenden Knoten bezeichnet.

Betrachte nun $G - k_{n+1}$. Die Kantenmenge ist

$$E - E(k_{n+1}) = \bigcup_{i=1}^{n+1} E(k_i) - E(k_{n+1}) = \bigcup_{i=1}^{n} E(k_i),$$

die somit insbesondere nicht leer ist. Es zeigt auch, dass $G - k_{n+1}$ in $\{k_1, \dots, k_n\}$ eine Kantenüberdeckung besitzt. Außerdem ist $G - k_{n+1}$ bis auf isolierte Knoten zusammenhängend. Um das einzusehen, seien v und v' zwei Knoten mit inzidenten Kanten, die deshalb jeweils auf einem der Kreise $k_1, \dots, k_n$ liegen, d.h. es gibt p und q mit $v \in V(k_p)$ und $v' \in V(k_q)$. Ohne Beschränkung der Allgemeinheit kann $p \leq q$ angenommen werden (sonst muss man die Rollen von v und v' vertauschen). Wenn $p = q$, dann teilen v und v' k_p in zwei Halbkreise, die v und v' durch Wege verbinden. Ist $p < q$, so existiert nach der obigen Voraussetzung ein Knoten v'' mit

$$v'' \in V(k_q) \cap \bigcup_{i=1}^{q-1} V(k_i),$$

d.h. $v'' \in V(k_r)$ für ein $r < q$. Induktiv[4] lässt sich nun voraussetzen, dass v und v'' durch einen Weg verbunden sind. Außerdem liegen v'' und v' auf k_q, so dass sie ebenfalls durch

[4] Die Induktion kann über das Maximum von p und q gemacht werden. Der Induktionsanfang für $p = q = 1$ ist im Fall $p = q$ gezeigt. Im Induktionsschluss sind p und r kleiner als q, so dass für das kleinere Maximum die Induktionsvoraussetzung gilt.

einen Weg (Halbkreis) verbunden sind. Beide Wege hintereinander ergeben einen Weg von v nach v', was den gewünschten Zusammenhang abschließend zeigt.

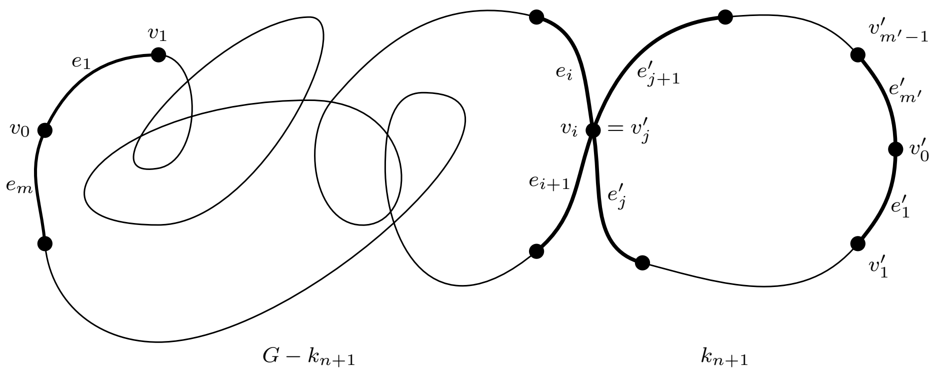
Mit anderen Worten gilt für $G - k_{n+1}$ die Induktionsvoraussetzung, so dass ein Eulerscher Kreis $v_0 e_1 \cdots e_m v_m = v_0$ in $G - k_{n+1}$ existiert. Insbesondere gilt $E - E(k_{n+1}) = \{e_1, \ldots e_m\}$. Sei $k_{n+1} = v'_0 e'_1 \cdots e'_m v'_m = v'_0$. Nach Voraussetzung gilt

$$V(k_{n+1}) \cap \bigcup_{i=1}^{n} V(k_i) \neq \emptyset,$$

so dass i, j existieren mit $v_i = v'_j$ und folgender Kreis gebildet werden kann:

$$v_0 e_1 \cdots e_i v_i = v'_j e'_{j+1} \cdots e'_m v'_m = v'_0 e'_1 \cdots e'_j v'_j = v_i e_{i+1} \cdots e_m v_m v_0.$$

Die Konstruktion lässt sich in einer Skizze veranschaulichen:



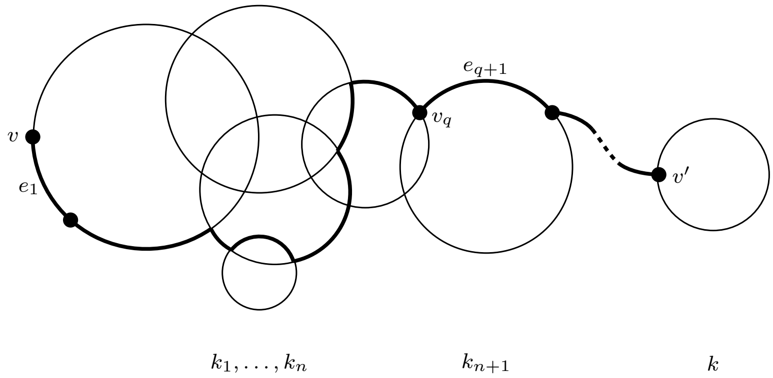
Das ist offensichtlich ein Eulerscher Kreis in G, denn jede Kante kommt genau einmal in je einem der beiden Teilkreise vor.

Es bleibt allerdings immer noch zu zeigen, dass die gemachte Voraussetzung keine Einschränkung darstellt. Sei dazu $k_1, \ldots, k_n$ eine Anordnung von Kreisen aus $\mathcal{K}$ mit der Eigenschaft

$$V(k_{m+1}) \cap \bigcup_{i=1}^{m} V(k_i) \neq \emptyset \qquad (*)$$

für $i = 1, \ldots, n-1$. Dabei ist $n = 1$ erlaubt, so dass jeder Kreis aus $\mathcal{K}$ eine Anordnung mit der gewünschten Eigenschaft ergibt. Wenn $\mathcal{K} = \{k_1, \ldots, k_n\}$, dann erfüllt $\mathcal{K}$ die Voraussetzung im Induktionsschluss. Ansonsten gibt es ein $k \in \mathcal{K}$ mit $k \notin \{k_1, \ldots, k_n\}$. Sei v ein Knoten auf einem der Kreise $k_1, \ldots, k_n$ und v' ein Knoten auf k. Da beide mit Kanten inzident sind, gibt es wegen des Zusammenhangs bis auf isolierte Knoten einen Weg $v = v_0 e_1 \cdots e_p v_p = v'$ von v nach v'. Sei $v_0 e_1 \cdots e_q v_q$ das längste Anfangsstück, das auf den Kreisen $k_1, \ldots, k_n$ liegt. Dann ist e_{q+1} außerhalb, und es existiert ein $k_{n+1} \in \mathcal{K}$ mit $e_{q+1} \in E(k_{n+1})$. Damit gilt $v_q \in V(k_{n+1}) \cap \bigcup_{i=1}^{n} V(k_i)$, so dass $k_1, \ldots, k_{n+1}$ eine Anordnung von Kreisen aus $\mathcal{K}$ mit der Eigenschaft $(*)$ für $i = 1, \ldots, n$ ist. Solange nicht alle Kreise angeordnet sind, lässt sich die Anordnung verlängern. Nach endlich vielen Verlängerungen sind somit alle Kreise geeignet angeordnet.

Auch dazu eine Skizze:



Insgesamt ist damit der Korrektheitsbeweis gelungen. Er ist nicht ganz einfach; aber gerade deshalb ist es wichtig zu wissen, dass es funktioniert. Man beachte, dass der Algorithmus zwei relativ simple Eigenschaften prüft, so dass er nur vergleichsweise bescheidene Erkenntnisse lieferte, wenn er kein korrekter Eulersch-Test wäre.

Kapitel 3

Färbungsprobleme

Färbungsprobleme sind eine wichtige Klasse von algorithmischen Problemen auf Graphen. Es werden in diesem Abschnitt explizit ungerichtete unmarkierte Graphen betrachtet, die aus einer Menge von Knoten und einer Menge von Kanten bestehen. Knoten repräsentieren individuelle Objekte irgendeiner Art. Kanten verbinden zwei Knoten und werden deshalb als zweielementige Teilmengen der Knotenmenge formalisiert. Eine Kante drückt eine Art Unverträglichkeit zwischen den verbundenen Knoten aus. Das Färbungsproblem ist dann ein Klassifizierungsproblem, bei dem die Knoten durch Färben so in Klassen eingeteilt werden, dass unverträgliche, also durch Kanten verbundene Knoten unterschiedlich gefärbt werden. Gesucht wird eine Färbung mit möglichst wenig Farben.

1. Ein (*unmarkierter, ungerichteter*) *Graph* $G = (V, E)$ besteht aus einer Menge V von *Knoten* und einer Menge E von *Kanten*, wobei jede Kante eine zweielementige Teilmenge von V ist.

   Für jede Kante $e \in E$ existieren also Knoten $v_1 \neq v_2$ mit $e = \{v_1, v_2\}$. Und wenn $\binom{V}{2}$ die Menge der zweielementigen Teilmengen von V bezeichnet, dann ist $E \subseteq \binom{V}{2}$.

2. Sei C eine Menge von k *Farben*. Sei $G = (V, E)$ ein Graph. Dann wird eine Abbildung $f \colon V \to C$ *k-Färbung* von G genannt, falls die Knoten jeder Kante $e = \{v_1, v_2\} \in E$ unterschiedlich gefärbt sind, d.h. $f(v_1) \neq f(v_2)$.

3. Sei $G = (V, E)$ ein Graph. Dann heißt $k \in \mathbb{N}$ die *Färbungszahl* von G, falls es eine k-Färbung von G gibt, aber keine $(k-1)$-Färbung. In diesem Fall heißt G k-färbbar.

Es ist leicht einzusehen, dass es für endliche Graphen immer eine minimale Färbung gibt, die aber im allgemeinen sehr aufwendig zu finden ist. Da nicht bekannt ist, ob es einen schnellen Algorithmus gibt, muss ein heuristisches Verfahren genügen, das nicht immer eine beste Lösung produziert. Die Erkenntnisse über Färbungen werden durch einfache Beobachtungen ergänzt.

4. Jeder Graph mit n Knoten lässt sich mit höchstens n Farben färben, denn jede injektive Abbildung von den Knoten in eine Farbenmenge ist eine Färbung.

5. Für einen Graphen mit n Knoten und für k Farben gibt es k^n Abbildungen von der Knoten- in die Farbenmenge und damit höchstens so viele Färbungen.

6. Damit erhält man einen einfachen Algorithmus zur Bestimmung einer minimalen Färbung:

   Färbung 1

   Teste für Graphen (mit mindestens einer Kante) und für wachsende k (beginnend mit 2 und spätestens endend mit der Knotenzahl) alle Abbildungen von der jeweiligen Knotenmenge in die Menge der Farben, ob sie Färbungen sind.

7. **Färbung 1** ist ein (mindestens) exponentieller Algorithmus, da er bei Eingabe von Graphen mit n Knoten und Färbungszahl $c > 2$ für $k = 2$ die 2^n Abbildungen von der Knoten- in die Farbenmenge erfolglos überprüft.

8. Es ist nicht bekannt, ob es einen polynomiellen Algorithmus gibt, der das Färbungsproblem löst. (Vgl. dazu später die Überlegungen zur NP-Vollständigkeit.)

9. Mit folgendem Verfahren erhält man in höchstens quadratisch vielen Schritten (bezogen auf die Knotenzahl) immer eine Färbung, die aber nicht minimal sein muss:

   Färbung 2

   Wiederhole, bis alle Knoten gefärbt sind:

   Wähle eine bisher nicht verwendete Farbe, und färbe damit jeden noch ungefärbten Knoten, falls er nicht mit einem Knoten dieser Farbe verbunden ist.

10. Genau die Graphen ohne Kanten sind 1-färbbar.

11. Der vollständige Graph K_n mit n Knoten, bei dem je zwei Knoten durch eine Kante verbunden sind, ist n-färbbar. Jeder andere Graph mit n Knoten ist mit weniger als n Farben färbbar.

12. Jeder Weggraph mit mindestens einer Kante ist 2-färbbar, indem die zwei Farben abwechselnd gesetzt werden.

13. Jeder Kreis mit gerade vielen Knoten ist 2-färbbar, indem die zwei Farben abwechselnd gesetzt werden.

14. Jeder Kreis mit ungerade vielen Knoten ist 3-färbbar, indem zwei Farben abwechselnd gesetzt werden und der letzte Knoten die dritte Farbe erhält.

15. Petri-Netze (bei denen man die Kantenrichtung ignoriert) sind 2-färbbar, denn alle Stellen können mit der einen, alle Transitionen mit der anderen Farbe gefärbt werden.

16. Jede ebene Landkarte ist höchstens 4-färbbar. Eine Landkarte kann als Graph gedeutet werden, indem man jedes Land zu einem Knoten macht und zwischen zwei Knoten eine Kante zieht, wenn die beiden Länder ein Stück Grenze gemeinsam haben. Diese Aussage wurde – bis heute nicht ganz unumstritten – 1977 von Appel und Haken [AH77] bewiesen. Bis dahin war sie als Vier-Farben-Problem oder Vier-Farben-Vermutung jahrzehntelang eines der offenen Probleme der Graphentheorie, an dem sich viele Wissenschaftlerinnen und Wissenschaftler versucht haben – und das trotz oder gerade wegen der vielen erfolglosen Bemühungen die Graphentheorie stark beeinflusst hat (vgl. [Aig84]).

Kapitel 4

Kürzeste Wege

Das Problem der kürzesten Wege gehört zu den anschaulichsten und wichtigsten algorithmischen Problemen auf Graphen mit einer immensen praktischen Bedeutung. Die Ausgangssituation besteht in einer Straßenkarte, deren Orte als Knoten gedeutet und Straßenverbindungen zwischen zwei Orten als Kanten interpretiert werden. Kanten werden außerdem mit der Entfernung zwischen den Orten beschriftet. Das Problem besteht dann darin, zwischen zwei Orten einen Weg zu finden, bei dem die Summe der Entfernungen aller durchlaufenen Kanten minimal ist. Manchmal interessiert auch nur diese kürzeste Entfernung und nicht so sehr ein konkreter Weg. Die Präzisierung und Formalisierung des Kürzeste-Wege-Problems werden für gerichtete Graphen gemacht, weil für die der Wegebegriff etwas einfacher ist. Außerdem können sie etwas flexibler angewendet werden, weil die Entfernung zwischen zwei Orten nicht unbedingt symmetrisch sein muss.

1. Ein *gerichtete Graph* $M = (V, E, s, t)$ besteht aus einer Menge V von *Knoten*, einer Menge E von *Kanten* und zwei Abbildungen $s\colon E \to V$ und $t\colon E \to V$, die jeder Kante $e \in E$ eine *Quelle* $s(e)$ und ein *Ziel* $t(e)$ zuordnen.

2. Ein *Weg* von v nach v' ($v, v' \in V$) der *Länge* n ist eine Kantenfolge $e_1 \cdots e_n$ ($n \geq 1$) mit $v = s(e_1)$, $v' = t(e_n)$ und $t(e_i) = s(e_{i+1})$ für $i = 1, \ldots, n-1$.

3. Aus technischen Gründen wird die leere Kantenfolge λ als (*leerer*) Weg von v nach v ($v \in V$) der Länge 0 betrachtet.

4. Die Menge aller Wege von v nach v' wird mit $PATH(v, v')$ bezeichnet, die Menge aller Wege von v nach v' der Länge n mit $PATH(v, v')_n$. Es gilt also:
$$PATH(v, v') = \bigcup_{n \in \mathbb{N}} PATH(v, v')_n.$$

5. Eine *Entfernung(sfunktion)* ist eine Abbildung $dist\colon E \to \mathbb{N}$.

   Statt natürlicher Zahlen kommen auch andere Wertebereiche wie die ganzen, rationalen und reellen Zahlen infrage.

6. Durch Addition der Einzelentfernungen lässt sich eine Entfernungsfunktion auf beliebige Kantenfolgen und damit insbesondere auf Wege fortsetzen: $dist\colon E^* \to \mathbb{N}$ definiert durch $dist(\lambda) = 0$ und $dist(e_1 \cdots e_n) = \sum_{k=1}^{n} dist(e_k)$ für alle $n \geq 1$ und $e_k \in E$, $k = 1, \ldots, n$.

7. Ein Weg $p_0 \in PATH(v, v')$ ist ein *kürzester Weg* von v nach v' (bzgl. $dist$), falls $dist(p_0) \leq dist(p)$ für alle $p \in PATH(v, v')$.

8. Ist $p_0 \in PATH(v, v')$ ein kürzester Weg, wird seine Entfernung mit $short(v, v')$ bezeichnet, d.h. $short(v, v') = dist(p_0)$.

9. Unter dem *Kürzeste-Wege-Problem* versteht man nun die Suche nach geeigneten Algorithmen zur Berechnung von $short(v, v')$ (beziehungsweise eines kürzesten Weges von v nach v' oder auch aller kürzesten Wege von v nach v') bei Eingabe eines beliebigen gerichteten Graphen, einer Entfernungsfunktion und zweier Knoten v und v' des Graphen.

Da die Menge der Wege in einem gerichteten Graphen unendlich sein kann, selbst wenn der Graph endlich ist, ist die Problembeschreibung im allgemeinen noch keine algorithmische Lösung. Denn konstruiert man alle Wege, bestimmt ihre Entfernung und greift darunter die kleinste heraus, wie es die Definition kürzester Wege verlangt, terminiert das Verfahren bei unendlich vielen Wegen nicht und findet deshalb nicht immer das richtige Ergebnis. Beachtet man aber, dass unendlich viele Wege in einem endlichen Graphen durch Kreise entstehen, Kreise Wege höchstens verlängern, Wege ohne Kreise höchstens so lang wie die Knotenzahl sein können und dass es bis zu einer bestimmten Länge nur endlich viele Wege gibt, erhält man einen ersten Algorithmus: Konstruiere alle einfachen Wege (das sind die Wege ohne Kreise), bestimme ihre Entfernung, greife darunter die kleinste heraus. Allerdings ist diese Lösung im ungünstigen Fall sehr aufwendig, weil ein Graph exponentiell viele einfache Wege haben kann (vgl. 1. Übungsblatt).

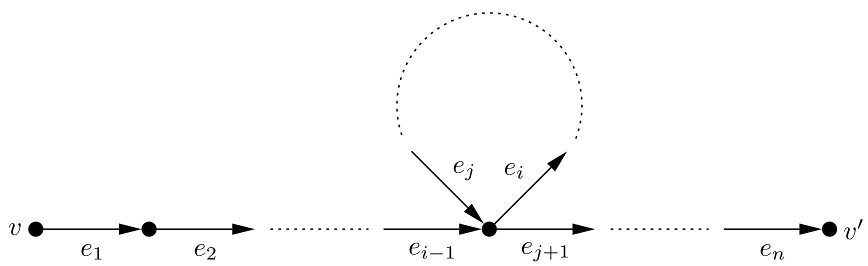
10. Ein Weg von v nach v der Länge $n \geq 1$ wird auch *Kreis* genannt.

11. Ein Weg $e_1 \cdots e_n$ heißt *einfach*, falls er keinen Kreis enthält, d.h. falls für keine Indizes $i < j$ der Teilweg $e_i \cdots e_j$ ein Kreis ist.

    Die Menge aller einfachen Wege von v nach v' wird mit $SIMPLE(v, v')$ bezeichnet.

12. Sei $p = e_1 \cdots e_n \in PATH(v, v')$ ein Weg, so dass der Teilweg $e_i \cdots e_j$ für ein Indexpaar $i \leq j$ ein Kreis ist. Dann ist $p(i, j) = e_1 \cdots e_{i-1} e_{j+1} \cdots e_n \in PATH(v, v')$ mit $dist(p(i, j)) \leq dist(p)$.

    Der Beweis ist sehr einfach, denn dass $p(i, j)$ auch ein Weg von v nach v' ist, ergibt sich unmittelbar aus der Situation:



Und dass die Entfernung dabei nicht größer wird, ist auch klar, weil die nichtnegativen Summanden $dist(e_i), \ldots, dist(e_j)$ im Vergleich von $dist(p)$ mit $dist(p(i,j))$ wegfallen:

$$dist(p(i,j)) = \sum_{k=1}^{i-1} dist(e_k) + \sum_{k=j+1}^{n} dist(e_k) \leq \sum_{k=1}^{n} dist(e_k) = dist(p).$$

13. Als Konsequenz für kürzeste Wege ergibt sich aus dieser Beobachtung:

    (i) Ist $p = e_1 \cdots e_n$ ein kürzester Weg und $e_i \cdots e_j$ mit $i \leq j$ ein Kreis, so gilt: $dist(e_i) = \cdots = dist(e_j) = 0$ und $dist(p(i,j)) = dist(p)$ (sonst wäre p kein kürzester Weg).

    (ii) Gibt es einen kürzesten Weg von v nach v', dann auch einen einfachen kürzesten Weg (da man alle Kreise aus einem kürzesten Weg herausnehmen kann).

14. Sei $\#V$ die Knotenzahl des Graphen M. Sei $p = e_1 \cdots e_n$ ein Weg mit $n \geq \#V$. Dann ist p nicht einfach.

    Das kann man sich klar machen, indem man die von p durchlaufenen Knoten $v_0, \ldots, v_n$ anschaut mit $v_0 = s(e_1)$ und $v_k = t(e_k)$ für $k = 1, \ldots, n$. Die Knotenfolge ist länger als die Knotenzahl, so dass mindestens ein Knoten doppelt vorkommen muss, d.h. $v_i = v_j$ für irgendwelche $i < j$. Dann muss aber der Teilweg $e_{i+1} \cdots e_j$ ein Kreis sein: $s(e_{i+1}) = v_i = v_j = t(e_j)$.

15. Sei $\#E$ die Kantenzahl des Graphen M. Dann gibt es $(\#E)^n$ Kantenfolgen der Länge n und damit höchstens $(\#E)^n$ Wege der Länge n. Insbesondere gibt es also höchstens endlich viele einfache Wege, da deren Länge durch die Knotenzahl beschränkt ist.

16. Die Überlegungen der Punkte 12 bis 15 können in einem Kürzeste-Wege-Algorithmus zusammengefasst werden:

    Kürzeste-Wege 1

    $$short_1(v, v') = \min(\{dist(p) \mid p \in SIMPLE(v, v')\})$$

    Dabei muss man die Menge $SIMPLE(v, v')$ irgendwie bilden. Beispielsweise könnte man alle Kantenfolgen kürzer als die Knotenzahl daraufhin überprüfen, ob sie einfache Wege bilden, und für diese dann die Entfernungen ausrechnen. Man erhält so eine endliche Menge natürlicher Zahlen, aus der das Minimum mit einem passenden Verfahren *min* herausgeholt werden kann.

    Der Algorithmus ist korrekt, denn wenn es einen Weg von v nach v' gibt, gilt nach den Punkten 12 bis 15: $short(v, v') = short_1(v, v')$. Gibt es jedoch keinen Weg von v nach v', also auch keinen einfachen und keinen kürzesten, dann ist $short(v, v')$ nicht definiert, während $short_1(v, v')$ das Minimum der leeren Menge liefert, also auch als nicht definiert angesehen werden kann.

Der Algorithmus kann allerdings sehr aufwendig sein, weil manche Graphen im Verhältnis zu ihrer Knotenzahl exponentiell viele einfache Wege besitzen, die möglicherweise alle durchprobiert werden. Ein schneller Algorithmus zur Berechnung der Entfernung kürzester Wege beziehungsweise einzelner kürzester Wege muss also verhindern, dass zu viele einfache oder kürzeste Wege einbezogen werden. Dazu bedarf es einer brauchbaren Idee. Ein Baustein zu einem schnellen Algorithmus ist die Tatsache, dass ein Weg p' von v nach $\bar{v}$ und ein Weg p'' von $\bar{v}$ nach v' zu einem Weg $p' \bullet p''$ von v nach v' zusammengesetzt werden können. Der eigentliche Schlüssel besteht in der Beobachtung der zwischendurch besuchten Knoten: Sind p' und p'' einfach und vermeiden zwischendurch die Knoten aus $\overline{\overline{V}}$, wobei $\overline{\overline{V}}$ eine Teilmenge der Gesamtknotenmenge mit $\bar{v} \in \overline{\overline{V}}$ ist, dann vermeidet $p' \bullet p''$ zwischendurch die Knoten aus $\overline{V} = \overline{\overline{V}} - \{\bar{v}\}$. Tatsächlich erweist sich ein kürzester einfacher Weg von v nach v', der $\overline{V}$ vermeidet, als einer, der auch schon $\overline{\overline{V}}$ vermeidet, oder der aus zwei Teilen zusammengesetzt ist, die beide kürzeste einfache Wege sind und $\overline{\overline{V}}$ vermeiden. So lassen sich kürzeste Wege, die weniger Knoten vermeiden, aus kürzesten Wegen konstruieren, die mehr Knoten vermeiden. Insgesamt erhält man so einen kubischen Algorithmus, der das *Kürzeste-Wege-Problem* löst.

17. Seien $p' = e_1 \cdots e_m \in PATH(v, \bar{v})$ und $p'' = e_{m+1} \cdots e_n \in PATH(\bar{v}, v')$. Dann ist $p' \bullet p'' = e_1 \cdots e_m e_{m+1} \cdots e_n \in PATH(v, v')$.

    Dieser Weg wird *Komposition* von p' und p'' genannt. Seine Länge ist gerade die Summe der Längen von p' und p''. Entsprechend ist seine Entfernung die Summe der Entfernungen von p' und p''.

18. Sei $p = e_1 \cdots e_n \in PATH(v, v')$. Dann bilden die *zwischendurch besuchten Knoten* $t(e_i)$ für $i = 1, \ldots, n-1$ die Menge $inter(p)$.

19. Die zwischendurch besuchten Knoten verhalten sich bezüglich der Komposition wie folgt: Sei $p' \in PATH(v, \bar{v})$ und $p'' \in PATH(\bar{v}, v')$. Dann wird $\bar{v}$ ein Zwischenknoten, und alle anderen Zwischenknoten bleiben erhalten, d.h. $inter(p' \bullet p'') = inter(p') \cup inter(p'') \cup \{\bar{v}\}$.

20. Ein Weg $p \in PATH(v, v')$ *vermeidet (zwischendurch)* die Knotenmenge $\overline{V} \subseteq V$, wenn kein Zwischenknoten von p in $\overline{V}$ liegt, wenn also $inter(p) \cap \overline{V} = \emptyset$.

    $SIMPLE(v, v', \overline{V})$ bezeichnet die Menge aller einfachen Wege von v nach v', die $\overline{V}$ vermeiden.

21. Ein Weg $p_0 \in SIMPLE(v, v', \overline{V})$ heißt *kürzester einfacher Weg* von v nach v', der $\overline{V}$ *vermeidet*, falls $dist(p_0) \leq dist(p)$ für alle $p \in SIMPLE(v, v', \overline{V})$. Seine Entfernung wird mit $short(v, v', \overline{V})$ bezeichnet.

    Die Menge der kürzesten einfachen Wege von v nach v', die $\overline{V}$ vermeiden, wird mit $SHORT(v, v', \overline{V})$ bezeichnet.

22. Sei $p_0 = e_1 \cdots e_n \in SHORT(v, v', \overline{V})$. Sei $\bar{v} \in V - \overline{V}$ und $\overline{\overline{V}} = \overline{V} \cup \{\bar{v}\}$. Dann ist $p_0 \in SHORT(v, v', \overline{\overline{V}})$, oder es existiert ein i_0 mit $t(e_{i_0}) = \bar{v}$, so dass $p_1 = e_1 \cdots e_{i_0} \in SHORT(v, \bar{v}, \overline{\overline{V}})$ und $p_2 = e_{i_0+1} \cdots e_n \in SHORT(\bar{v}, v', \overline{\overline{V}})$.

    Das ist noch relativ leicht einzusehen. Nach Voraussetzung ist $inter(p_0) \cap \overline{V} = \emptyset$.

    Betrachte zuerst den Fall, dass $\bar{v} \notin inter(p_0)$. Dann ist $p_0 \in SIMPLE(v, v', \overline{\overline{V}})$. Außerdem gilt offenbar $SIMPLE(v, v', \overline{\overline{V}}) \subseteq SIMPLE(v, v', \overline{V})$, denn je mehr Knoten man vermeiden muss, desto weniger Wege kann es geben. Der gegebene Weg p_0 ist aber bereits ein kürzester in der größeren Menge, also erst recht in der kleineren.

Es bleibt der Fall $\bar{v} \in \mathit{inter}(p_0)$, was aber gerade die Existenz eines $i_0 \in [n-1]$ bedeutet mit $t(e_{i_0}) = \bar{v}$. Man kann dann p_0 in $\bar{v}$ durchschneiden und erhält die Wege p_1 und p_2, die offensichtlich einfach sind und $\overline{\overline{V}}$ vermeiden, weil sonst p_0 nicht einfach wäre. Schließlich bleibt zu zeigen, dass p_1 kürzester Weg ist. Angenommen, es gäbe $\bar{p}_1 \in \mathit{SHORT}(v, \bar{v}, \overline{\overline{V}})$ mit $\mathit{dist}(\bar{p}_1) < \mathit{dist}(p_1)$. Dann kann man $\bar{p}_1 \bullet p_2$ als Weg von v nach v' bilden, der $\overline{V}$ vermeidet. Außerdem gilt $\mathit{dist}(\bar{p}_1 \bullet p_2) = \mathit{dist}(\bar{p}_1) + \mathit{dist}(p_2) < \mathit{dist}(p_1) + \mathit{dist}(p_2) = \mathit{dist}(p_1 \bullet p_2) = \mathit{dist}(p_0)$. Das ist ein Widerspruch zu $p_0 \in \mathit{SHORT}(v, v', \overline{V})$, so dass $p_1 \in \mathit{SHORT}(v, \bar{v}, \overline{\overline{V}})$ sein muss. Genauso zeigt man, dass $p_2 \in \mathit{SHORT}(\bar{v}, v', \overline{\overline{V}})$ ist.

23. Für die Entfernung kürzester Wege, die Knoten vermeiden, ergeben sich folgende Beziehungen:

    - $\mathit{short}(v, v', \overline{V}) = \mathit{short}(v, v', \overline{\overline{V}})$ oder $\mathit{short}(v, v', \overline{V}) = \mathit{short}(v, \bar{v}, \overline{\overline{V}}) + \mathit{short}(\bar{v}, v', \overline{\overline{V}})$,
    - $\mathit{short}(v, v', \overline{V}) \leq \mathit{short}(v, v', \overline{\overline{V}})$,
    - $\mathit{short}(v, v', \overline{V}) \leq \mathit{short}(v, \bar{v}, \overline{\overline{V}}) + \mathit{short}(\bar{v}, v', \overline{\overline{V}})$.

    Falls $\mathit{SHORT}(v, v', \overline{V})$ nicht leer ist und damit $\mathit{short}(v, v', \overline{V})$ eine natürliche Zahl, ergeben sich die Beziehungen aus dem vorigen Punkt und seiner Begründung. Bleibt der Fall $\mathit{short}(v, v', \overline{V}) = \infty$. Dann müssen aber $\mathit{SHORT}(v, v', \overline{\overline{V}})$ und $\mathit{SHORT}(v, \bar{v}, \overline{\overline{V}})$ oder $\mathit{SHORT}(\bar{v}, v', \overline{\overline{V}})$ ebenfalls leer sein, weil sonst ein Weg von v nach v' existierte, der $\overline{V}$ vermeidet. Und damit gilt $\mathit{short}(v, v', \overline{\overline{V}}) = \infty$ und $\mathit{short}(v, \bar{v}, \overline{\overline{V}}) + \mathit{short}(\bar{v}, v', \overline{\overline{V}}) = \infty$, so dass die obigen Beziehungen immer noch gelten.

24. Die Beziehungen des vorigen Punktes lassen sich in einer Gleichung zusammenfassen:

    $$\mathit{short}(v, v', \overline{V}) = \min(\mathit{short}(v, v', \overline{\overline{V}}), \mathit{short}(v, \bar{v}, \overline{\overline{V}}) + \mathit{short}(\bar{v}, v', \overline{\overline{V}})).$$

25. Damit ist der wesentliche Schritt eines Algorithmus für kürzeste Wege konstruiert, bei dem rekursiv die Menge der vermiedenen Knoten vergrößert wird. Nur die Initialisierung und die Abbruchbedingungen kommen hinzu:

    Kürzeste-Wege 2

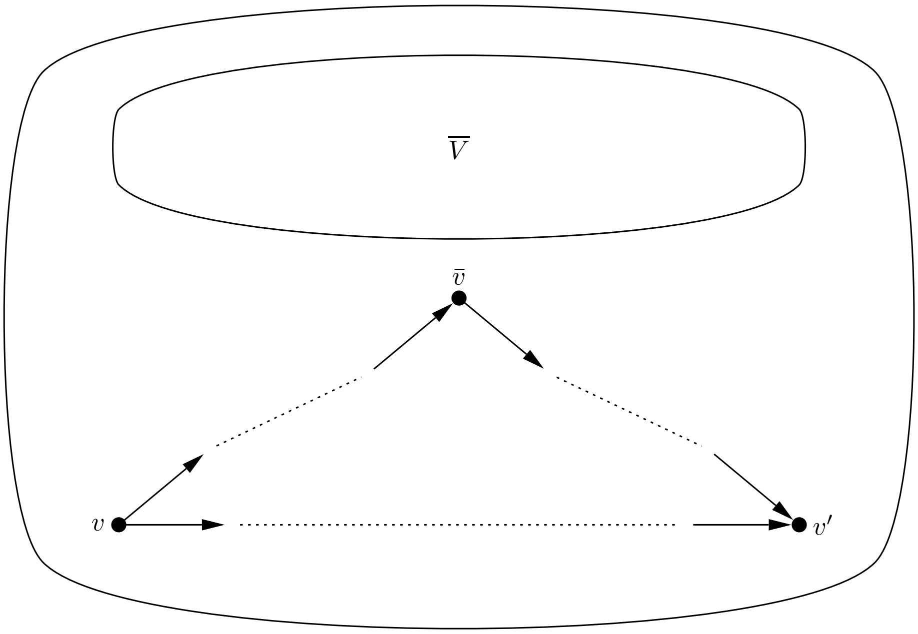
    $$\begin{aligned}
    &\mathit{short}_2(v, v') = \mathit{short}_2(v, v', \emptyset),\\
    &\mathit{short}_2(v, v', \overline{V}) = \min(\mathit{short}_2(v, v', \overline{\overline{V}}), \mathit{short}_2(v, \bar{v}, \overline{\overline{V}}) + \mathit{short}_2(\bar{v}, v', \overline{\overline{V}}))\\
    &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{für } v \neq v',\\
    &\mathit{short}_2(v, v, \overline{V}) = 0,\\
    &\mathit{short}_2(v, v', V) = \min(\{\mathit{dist}(e) \mid e \in E,\ s(e) = v,\ t(e) = v'\}) \qquad \text{für } v \neq v'.
    \end{aligned}$$

    Die erste Zeile ist klar, denn kürzeste Wege, die nichts vermeiden müssen, sind allgemeine kürzeste Wege. Die zweite Zeile ist nach dem vorigen Punkt korrekt. Der leere Weg ist immer ein kürzester Weg von v nach v, der zwischendurch keine Knoten besucht. Das erklärt die dritte Zeile. Die kürzesten Wege zwischen zwei Knoten, die zwischendurch alle Knoten vermeiden, können nur die Kanten zwischen den beiden Knoten mit der kleinsten Entfernung sein, was die vierte Zeile ergibt.

26. Der Algorithmus $short_2$ hat einen kubischen Aufwand im Vergleich zur Knotenzahl.

    In der ersten und vierten Zeile sind quadratisch viele Werte zu berechnen. Bei der ersten Zeile ist pro Wert ein anderer zu bestimmen, bei der vierten Zeile nimmt man an, dass der Wert bekannt ist. Bezüglich der zweiten Zeile variieren die ersten beiden Argumente von $short_2$ über alle Knoten, das dritte Argument muss auch nur, beginnend mit der leeren Menge, alle Knoten zum Vermeiden einmal durchlaufen, obwohl es prinzipiell für alle Teilmengen der Knotenmenge definiert ist. Also sind kubisch viele Werte zu berechnen. Um einen davon zu ermitteln, müssen drei andere Werte ermittelt, zwei davon addiert und ein Minimum von zwei Zahlen bestimmt werden. Da die drei Werte als vorher berechnet angenommen werden können, lässt sich das alles in konstanter Zeit bewerkstelligen, so dass der Aufwand höchstens kubisch ist. Bei der dritten Zeile ist analog quadratisch oft eine Konstante zu schreiben. Insgesamt ergibt sich bis auf einen konstanten Faktor ein kubischer Aufwand.

27. Die folgende Illustration stellt die in Punkt 22 beschriebene Situation graphisch dar, die den zentralen Rekursionsschritt des Algorithmus ergibt. Ein kürzester Weg von v nach v', der $\overline{V}$ vermeidet, besucht entweder einen weiteren Bezugsknoten $\bar{v}$ oder vermeidet diesen zusätzlich.



Kapitel 5

Minimale aufspannende Bäume

Das Problem der minimalen aufspannenden Bäume ist mit dem Kürzeste-Wege-Problem verwandt, allerdings geht es nicht mehr um Verbindungen zwischen zwei Knoten, sondern zwischen allen Knoten gleichzeitig. Die Ausgangssituation ist ein ungerichteter Graph (ohne Schleifen und Mehrfachkanten) mit einer Kostenfunktion. In den Anwendungen werden dabei die Knoten oft als lokale Sende- und Empfangsstationen oder Prozessoren gedeutet und die Kanten als direkte Nachrichtenkanäle. Die Kosten mögen dann konkret so etwas wie Mietkosten o.ä. repräsentieren. Die Aufgabe besteht darin, in einem solchen Kommunikationsnetz möglichst „kostengünstig“ Nachrichten von einem Knoten zu allen anderen senden zu können. Minimiert werden soll dabei die Kostensumme über alle benutzten Kanten.

Das Problem macht nur Sinn, wenn der Ausgangsgraph zusammenhängend ist, d.h. wenn je zwei Knoten durch Wege verbunden sind, weil sonst Nachrichten nicht von überall her nach überall hin geschickt werden können. Dann bildet aber der Ausgangsgraph einen ersten Lösungsversuch, bei dem alle Kanten als benutzt angesehen werden. Löscht man nun Kanten, wird die Lösung besser, wobei man aufpassen muss, dass der Graph beim Löschen von Kanten nicht seinen Zusammenhang verliert. Bei diesem Prozess entstehen Teilgraphen des ursprünglichen Graphen, die die ursprüngliche Knotenmenge behalten und zusammenhängend sind. Der Prozess muss enden, wenn keine Kante mehr gelöscht werden kann, ohne dass der Graph in mehrere unzusammenhängende Komponenten zerfällt. Zusammenhängende Graphen, die zerfallen, sobald eine beliebige Kante entfernt wird, sind einprägsamer Bäume. Teilgraphen mit den ursprünglichen Knoten als Knotenmenge heißen aufspannend. Das skizzierte Verfahren produziert also aufspannende Bäume, die sogar minimal werden, wenn man in jedem Schritt die größtmögliche Kante entfernt.

1. Behandelt werden in diesem Kapitel ungerichtete Graphen ohne Schleifen und Mehrfachkanten, d.h. ohne Beschränkung der Allgemeinheit Systeme der Form $G = (V, E)$ mit $E \subseteq \binom{V}{2}$. Betrachtet man diese Inklusion als Aufhängung, sind das ungerichtete Graphen im Sinne von Kapitel 2.

2. Ein Graph $G = (V, E)$ ist *zusammenhängend*, wenn es je einen Weg von v nach v' für alle $v, v' \in V$ gibt.

3. Ein Graph $G' = (V', E')$ ist ein *Teilgraph* von G (in Zeichen: $G' \subseteq G$), falls $V' \subseteq V$ und $E' \subseteq E$. G' heißt *aufspannend*, falls $V' = V$.

Aus G entsteht durch Löschen von Knoten und Kanten ein Teilgraph, wenn mit jedem gelöschten Knoten alle daran hängenden Kanten gelöscht werden. Durch Löschen von Kanten entsteht ein aufspannender Teilgraph.

4. Eine Kante $e \in E$ heißt *Brücke* eines zusammenhängenden Graphen $G = (V, E)$, falls der Teilgraph $G - e = (V, E - \{e\})$ nicht zusammenhängt. Bezeichne $BRIDGE(G)$ die Menge aller Brücken von G.

5. Ein zusammenhängender Graph G, bei dem alle Kanten Brücken sind: $BRIDGE(G) = E$, wird *Baum* genannt. Ein Baum, der aufspannender Teilgraph ist, wird *aufspannender Baum* genannt und die Menge aller aufspannenden Bäume eines zusammenhängenden Graphen G mit $SPANTREE(G)$ bezeichnet.

6. Sei $G = (V, E)$ ein zusammenhängender Graph und $cost\colon E \to \mathbb{N}$ eine Kostenfunktion. Ein aufspannender Baum $B \in SPANTREE(G)$ ist *minimal* (bezüglich *cost*), falls $cost(B) \leq cost(B')$ für alle $B' \in SPANTREE(G)$.

   Dabei sind die *Kosten* für $G' \subseteq G$ mit $G' = (V', E')$ gegeben durch

   $$cost(G') = \sum_{e' \in E'} cost(e').$$

7. Der folgende Algorithmus konstruiert mit polynomiellem Aufwand zu jedem zusammenhängenden Graphen und zu jeder Kostenfunktion einen minimalen aufspannenden Baum:

   min-spantree 1

Eingabe:	$G = (V, E)$ zusammenhängend, $cost\colon E \to \mathbb{N}$.
Verfahren:	Solange $E - BRIDGE(G) \neq \emptyset$: (1) wähle $e \in E - BRIDGE(G)$ mit $cost(e) \geq cost(e')$ für alle $e' \in E - BRIDGE(G)$, (2) setze $E = E - \{e\}$ und $G = G - e$.

   Wenn man die Eingabe nicht verändern möchte, muss man eine Graphvariable mit G initialisieren und auf der dann das Verfahren laufen lassen.

   Das Verfahren terminiert, weil in jedem Schritt eine Kante gelöscht wird, von denen es nur endlich viele gibt. In jedem Schritt wird jede Kante daraufhin überprüft, ob sie auf einem Kreis liegt und damit keine Brücke ist. Außerdem müssen alle Kanten der Größe nach sortiert werden. Beides geht in polynomieller Zeit. Nach Konstruktion wird nie eine Brücke entfernt, so dass alle konstruierten Teilgraphen zusammenhängend sind. Da die Knotenmenge unverändert bleibt, sind alle Teilgraphen aufspannend. Abgebrochen wird, wenn alle Kanten Brücken sind, also mit einem aufspannenden Baum. Bleibt die Minimalität zu zeigen. Darauf wird allerdings bei diesem Algorithmus verzichtet, weil es einen ähnlichen Algorithmus gibt, bei dem das etwas einfacher gezeigt werden kann und der darüber hinaus auch weniger Aufwand hat.

Einen noch etwas besseren Algorithmus, der von Kruskal 1956 vorgeschlagen wurde, erhält man, wenn man umgekehrt vorgeht und, beginnend mit der leeren Menge, Kanten zufügt, die möglichst geringe Kosten haben und mit den schon ausgewählten keine Kreise bilden.

8. Der Algorithmus von Kruskal konstruiert mit polynomiellem Aufwand für einen zusammenhängenden Graphen mit Kostenfunktion folgendermaßen einen minimalen aufspannenden Baum:

min-spantree 2

Eingabe: $G = (V, E)$ zusammenhängend,
$cost\colon E \to \mathbb{N}$

Verfahren: (1) Sortiere $E = \{e_1, \ldots, e_m\}$, so dass
$cost(e_1) \leq \cdots \leq cost(e_m)$;
(2) $E_0 := \emptyset$;
für $i = 1, \ldots, m$:
$$E_i := \begin{cases} E_{i-1} \cup \{e_i\} & \text{falls } (V, E_{i-1} \cup \{e_i\}) \text{ kreisfrei ist und} \\ E_{i-1} & \text{sonst.} \end{cases}$$

Ausgabe: $B_m = (V, E_m)$.

Nach Konstruktion ist B_m ein aufspannender und kreisfreier Teilgraph. B_m ist auch zusammenhängend. Denn gäbe es zwei Knoten v_1 und v_2, die nicht durch einen Weg verbunden wären, dann läge auf einem Weg p, der v_1 und v_2 in G verbindet, eine Kante e_j, die nicht in E_m liegt. Das ist aber nur dann der Fall, wenn in $(V, E_{j-1} \cup \{e_j\})$ ein Kreis und damit auch ein einfacher Kreis liegt. Da (V, E_{j-1}) keinen Kreis enthält, liegt e_j auf diesem Kreis. Löscht man e_j von diesem Kreis, bleibt ein Weg in $(V, E_{j-1}) \subseteq B_m$ übrig, der e_j in p ersetzen kann. Der Weg p lässt sich also nach und nach in einen Weg zwischen v_1 und v_2 in B_m verwandeln im Widerspruch zur Wahl von v_1 und v_2. B_m besteht als kreisfreier, zusammenhängender Graph aber nur aus Brückenkanten, so dass B_m ein aufspannender Baum ist.

Bleibt die Minimalität zu zeigen. Als Baum enthält E_m gerade $n - 1$ Kanten für $n = \#V$, was gesondert bewiesen wird. Seien dies $e_{i_1}, \ldots, e_{i_{n-1}}$ mit $i_1 < \cdots < i_{n-1}$, die zu $\bar{e}_1, \ldots, \bar{e}_{n-1}$ umbenannt werden. Sei E_B die Kantenmenge eines beliebigen aufspannenden Baumes B. Dann enthält die Differenz $E_m - E_B$ alle algorithmisch gewählten Kanten, die nicht im Vergleichsbaum vorkommen. Gezeigt werden soll, dass $cost(B_m) \leq cost(B)$, so dass sich B_m als minimal unter den aufspannenden Bäumen erweist. Der Beweis wird über die Zahl $\#(E_m - E_B)$ der abweichenden Kanten geführt.

IA: $\#(E_m - E_B) = 0$, d.h. $E_m \subseteq E_B$ und damit $cost(B_m) \leq cost(B)$, da kein Summand negativ ist.

I.V.: Die Behauptung gelte für $\#(E_m - E_B) = k \geq 0$.

I.S.: Betrachte E_B mit $\#(E_m - E_B) = k+1$. Dann gibt es ein kleinstes l mit $\bar{e}_1, \ldots, \bar{e}_{l-1} \in E_B$, aber $\bar{e}_l \notin E_B$. Sei $\bar{e}_l = \{v_1, v_2\}$. Da B ein aufspannender Baum ist, enthält B einen Weg $p(\bar{e}_l)$, der die Knoten v_1 und v_2 verbindet. Mindestens eine Kante e_0 dieses Weges liegt nicht in E_m, weil sonst $p(\bar{e}_l)\bar{e}_l$ einen Kreis in B_m bildete. B_m aber ist kreisfrei. Die Kante e_0 bildet insbesondere mit den Kanten $\bar{e}_1, \ldots, \bar{e}_l$ keinen Kreis. Also liegt die Kante e_0 in der Sortierung am Anfang des Algorithmus hinter $\bar{e}_l$, woraus sich $cost(\bar{e}_l) \leq cost(e_0)$ ergibt.

Betrachte nun $B' = (V, (E_B - \{e_0\}) \cup \{\bar{e}_l\})$. Das ist nach Konstruktion ein aufspannender Teilgraph. B' ist außerdem zusammenhängend. Denn zwischen zwei Knoten

v und v' gibt es in B einen Weg, der entweder e_0 nicht enthält und damit auch in B' liegt oder bei dem e_0 ersetzt werden kann durch den Weg p_0, der entsteht, wenn man aus dem Kreis $p(\bar{e}_l)\bar{e}_l$ die Kante e_0 entfernt, so dass ein Weg in B' entsteht. B' ist schließlich auch kreisfrei. Denn gäbe es einen Kreis, dann läge der widersprüchlich auch schon in B oder ginge durch $\bar{e}_l$. Der Kreis ohne $\bar{e}_l$ ist dann ein Weg zwischen v_1 und v_2 in B, der e_0 nicht enthält. Da B mit $p(\bar{e}_l)$ noch einen Weg zwischen v_1 und v_2 enthält, auf dem e_0 liegt, hätte man insgesamt auch einen Kreis in B im Widerspruch zur Baumeigenschaft. B' ist also insgesamt ein aufspannender Baum. Außerdem gilt: $\#(E_m - ((E_B - \{e_0\}) \cup \{\bar{e}_l\})) = \#(E_m - E_B) - 1 = k$, da $e_0 \notin E_m$ und $\bar{e}_l \in E_m - E_B$. Nach IV ergibt sich $cost(B_m) \leq cost(B')$. Wegen

$$\begin{aligned} cost(B') &= cost((E_B - \{e_0\}) \cup \{\bar{e}_l\}) \\ &= cost(E_B) - cost(e_0) + cost(\bar{e}_l) \\ &\leq cost(E_B) \\ &= cost(B), \end{aligned}$$

was sich aus der oben gezeigten Beziehung $cost(\bar{e}_l) \leq cost(e_0)$ ergibt, folgt daraus wie gewünscht $cost(B_m) \leq cost(B)$.

Damit ist die Korrektheit gezeigt. Zum Aufwand lässt sich feststellen, dass das Sortieren der Kanten bekanntlich in $O(m \log m)$ Schritten möglich ist. Danach muss für die m Kanten e_i ermittelt werden, ob in (V, E_{i-1}) ein Weg die beiden Knoten von e_i verbindet, was mit dem Dijkstra-Algorithmus höchstens $O(n^2)$ Schritte benötigt. Insgesamt also $O(\max(m \log m, mn^2))$. Dabei kann man noch beachten, dass $m \leq \binom{n}{2}$ ist, so dass bei dieser Abschätzung n^4 als Schranke herauskommt. Wenn man das Ganze noch etwas geschickter verwaltet, lässt sich die Schranke sogar auf $n^2 \log n$ drücken. Darauf wird hier aber verzichtet.

Kapitel 6

Maximale Flüsse

Beim Flussproblem wird ein gerichteter Graph betrachtet, der ein Rohrleitungssystem oder ein Verkehrsnetz o. ä. darstellt. Durch jede Kante kann ein (Flüssigkeits- oder Verkehrs-)Fluss bis zu einer vorgegebenen Kapazität fließen. Es gibt zwei ausgezeichnete Knoten, Quelle und Senke, zwischen denen der Fluss von der Quelle zur Senke einen Durchsatz erbringen soll. Dabei wird als Durchsatz die Flussquantität bezeichnet, die bei der Quelle abfließt und nicht wieder zufließt, beziehungsweise die bei der Senke zufließt und nicht wieder abfließt. Von allen anderen Knoten wird angenommen, dass genauso viel abfließt, wie zufließt. Die Aufgabe besteht nun darin, einen Fluss mit möglichst großem Durchsatz zu finden. Formal gefasst, sieht das Flussproblem folgendermaßen aus:

1. Ein *Kapazitätsnetz* $N = (G, A, B, cap)$ besteht aus einem gerichteten Graphen $G = (V, E, s, t)$, zwei ausgezeichneten Knoten $A, B \in V$, wobei A *Quelle* und B *Senke* genannt wird, sowie einer *Kapazität(sfunktion)* $cap\colon E \to \mathbb{N}$.

2. Unter einem *Fluss* versteht man dann eine Funktion $flow\colon E \to \mathbb{N}$, die folgenden Eigenschaften genügt:

   (i) $flow(e) \leq cap(e)$ für alle $e \in E$,

   (ii) $\sum\limits_{s(e)=v} flow(e) = \sum\limits_{t(e')=v} flow(e')$ für alle $v \in V$ mit $A \neq v \neq B$.

3. Es gilt dann: $\sum\limits_{s(e)=A} flow(e) - \sum\limits_{t(e')=A} flow(e') = \sum\limits_{t(e)=B} flow(e) - \sum\limits_{s(e')=B} flow(e')$.

   Diese Zahl wird *Durchsatz* genannt und mit $val(flow)$ bezeichnet. Der Beweis, dass die beiden Differenzen gleich sind, wird in Punkt 9 geführt.

4. Bezeichnet man die Summe $\sum\limits_{t(e)=v} flow(e)$ der Flüsse von Kanten, die in den Knoten v münden, mit $inflow(v)$ und analog $outflow(v) = \sum\limits_{s(e)=v} flow(e)$, dann lässt sich die zweite Flussbedingung einprägsamer formulieren als

$$inflow(v) = outflow(v) \text{ für alle } v \in V \text{ mit } A \neq v \neq B.$$

Und für den Durchsatz gilt dann:

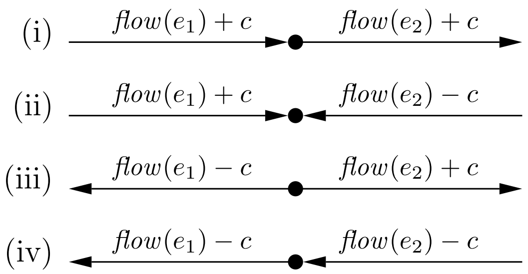
$$val(\mathit{flow}) = \mathit{outflow}(A) - \mathit{inflow}(A) = \mathit{inflow}(B) - \mathit{outflow}(B).$$

Manchmal wird für Kapazitätsnetze in der Literatur verlangt, dass keine Kanten in A ankommen und keine von B weggehen. In diesem Fall gilt für den Durchsatz einfach:

$$val(\mathit{flow}) = \mathit{outflow}(A) = \mathit{inflow}(B).$$

Auf diese Vereinfachung wird verzichtet, weil dann A und B leichter variiert werden können.

5. Am Knoten A lässt sich ein Fluss $\mathit{flow}\colon E \to \mathbb{N}$ verbessern, wenn es gelingt, den Fluss einmündender Kanten zu verkleinern und den ausgehender zu vergrößern, und umgekehrt am Knoten B. Wird jedoch der Fluss einer Kante geändert und ist ihre Quelle oder ihr Ziel von A oder B verschieden, dann muss zur Erhaltung der zweiten Flussbedingung auch der Fluss einer im selben Knoten anhängenden Kante geändert werden usw. Dabei treten vier Fälle auf:



Außerdem muss die erste Flussbedingung beachtet werden, nach der höchstens die Differenz zwischen Kapazität und aktuellem Fluss zur Erhöhung zur Verfügung steht, während der aktuelle Fluss höchstens auf Null gesenkt werden kann.

Nach diesen Überlegungen gelingt eine Flussverbesserung, wenn man einen *aufsteigenden Pfad* von A nach B findet.

Sei für $e \in E$ $\bar{e}$ eine sogenannte *Rückwärtskante*, sei $\bar{E} = \{\bar{e} \mid e \in E\}$, und bezeichne $\bar{G}$ den um die Rückwärtskanten erweiterten Graphen G, d.h. $\bar{G} = (V, E \cup \bar{E}, \bar{s}, \bar{t})$ mit $\bar{s}(e) = s(e)$ und $\bar{s}(\bar{e}) = t(e)$ sowie $\bar{t}(e) = t(e)$ und $\bar{t}(\bar{e}) = s(e)$ für alle $e \in E$. Dann versteht man unter einem *aufsteigenden Pfad* von v nach v' bzgl. *flow* einen einfachen Weg $\hat{e}_1 \cdots \hat{e}_n$ von v nach v' in $\bar{G}$ mit $\mathit{cap}(\hat{e}_i) - \mathit{flow}(\hat{e}_i) > 0$ für $\hat{e}_i \in E$ und $\mathit{flow}(\hat{e}_i) > 0$ für $\hat{e}_i \in \bar{E}$ $(i = 1, \ldots, n)$.

Sei $\hat{e}_1 \cdots \hat{e}_n$ ein aufsteigender Pfad von A nach B. Sei min die kleinste unter den Zahlen $\mathit{cap}(\hat{e}_i) - \mathit{flow}(\hat{e}_i)$ für $\hat{e}_i \in E$ und $\mathit{flow}(\hat{e}_i)$ für $\hat{e}_i \in \bar{E}$ $(i = 1, \ldots, n)$. Dann wird durch $\mathit{flow}'\colon E \to \mathbb{N}$ mit

$$\mathit{flow}'(e) = \begin{cases} \mathit{flow}(e) + \mathit{min} & \text{falls } e = \hat{e}_{i_0} \text{ für ein } i_0 \\ \mathit{flow}(e) - \mathit{min} & \text{falls } \bar{e} = \hat{e}_{j_0} \text{ für ein } j_0 \\ \mathit{flow}(e) & \text{sonst} \end{cases}$$

ein Fluss definiert, für dessen Durchsatz gilt:

$$val(\mathit{flow}') = val(\mathit{flow}) + \mathit{min}.$$

6. Jeder beliebige initiale Fluss, wofür z.B. der *Nullfluss* $zero\colon E \to \mathbb{N}$ mit $zero(e) = 0$ für alle $e \in E$ mit dem Durchsatz $val(zero) = 0$ verwendet werden kann, lässt sich durch wiederholte Verbesserung mit Hilfe aufsteigender Pfade zu einem Fluss verbessern, der keine aufsteigenden Pfade mehr von A nach B besitzt. Das ist der Algorithmus von Ford nach Fulkerson aus dem Jahre 1956 (siehe z.B. [Jun90]).

   Der Algorithmus terminiert, weil in jedem Schritt der Durchsatz ganzzahlig steigt, der nach oben z.B. durch die Kapazitätssumme aller Kanten mit A als Quelle beschränkt ist (vgl. Punkt 8).

   Der Algorithmus liefert darüber hinaus immer einen Fluss mit maximalem Durchsatz unter allen Flüssen des gegebenen Kapazitätsnetzes. Das ergibt sich aus dem folgenden Resultat.

7. Ein Fluss *flow* ist genau dann maximal, wenn er keinen aufsteigenden Pfad von A nach B besitzt.

   Die eine Richtung des Beweises ist einfach: Sei *flow* ein maximaler Fluss. Gäbe es nun einen aufsteigenden Pfad von A nach B, ließe sich der Fluss im Widerspruch zu seiner Maximalität gemäß Punkt 5 noch verbessern. Also gibt es keinen aufsteigenden Pfad von A nach B.

   Die Umkehrung beruht auf dem Konzept des Schnittes und seinen Eigenschaften, was im nächsten Punkt näher betrachtet wird.

8. Ein *Schnitt* (S,T) besteht aus zwei Knotenmengen $S,T \subseteq V$ mit $S \cup T = V$, $S \cap T = \emptyset$ sowie $A \in S$ und $B \in T$. Nach Definition ist T immer das Komplement von S und umgekehrt.

   Für einen Fluss *flow* heißt die Flusssumme der Kanten von S nach T vermindert um die Flusssumme der Kanten von T nach S *Fluss* des Schnittes:

   $$flow(S,T) = \sum_{\substack{s(e)\in S\\ t(e)\in T}} flow(e) - \sum_{\substack{t(e)\in S\\ s(e)\in T}} flow(e).$$

   Die Kapazitätssumme der Kanten von S nach T heißt *Kapazität* des Schnittes:

   $$cap(S,T) = \sum_{\substack{s(e)\in S\\ t(e)\in T}} cap(e).$$

   Alles was fließt, muss durch die Kanten zwischen S und T, so dass der Fluss eines Schnittes mit dem Durchsatz übereinstimmt und dieser wiederum nach oben durch die Kapazität des Schnittes beschränkt sein muss. Formaler gefasst, gilt:

   $$val(flow) = flow(S,T) \leq cap(S,T).$$

   Um das einzusehen, betrachte man die Summe aller Differenzen von Ausflüssen und Einflüssen der Knoten in S. Einerseits ist sie gerade der Durchsatz, weil die Differenzen für alle Knoten, die von A verschieden sind, nach der zweiten Flusseigenschaft Null sind:

   $$\sum_{v\in S}(outflow(v) - inflow(v)) = outflow(A) - inflow(A) = val(flow).$$

Andererseits kann man die Summanden ordnen nach den Flüssen der Kanten, die Quelle und Ziel in S haben, und denen, die nur ihre Quelle oder nur ihr Ziel in S haben. Da dabei die Flüsse von Kanten mit Quelle und Ziel in S sowohl mit positivem als auch negativem Vorzeichen vorkommen, bleibt der Fluss des Schnittes übrig:

$$\begin{aligned}
\sum_{v\in S}(\mathit{outflow}(v) - \mathit{inflow}(v)) &= \sum_{v\in S}\Big(\sum_{s(e)=v}\mathit{flow}(e) - \sum_{t(e)=v}\mathit{flow}(e)\Big)\\
&= \sum_{s(e)\in S}\mathit{flow}(e) - \sum_{t(e)\in S}\mathit{flow}(e)\\
&= \sum_{\substack{s(e)\in S\\ t(e)\in T}}\mathit{flow}(e) + \sum_{\substack{s(e)\in S\\ t(e)\in S}}\mathit{flow}(e) - \sum_{\substack{t(e)\in S\\ s(e)\in S}}\mathit{flow}(e) - \sum_{\substack{t(e)\in S\\ s(e)\in T}}\mathit{flow}(e)\\
&= \mathit{flow}(S,T).
\end{aligned}$$

Dass dieser Fluss des Schnittes unterhalb der Kapazität des Schnittes liegt, ist wegen der ersten Flusseigenschaft klar:

$$\begin{aligned}
\mathit{flow}(S,T) &= \sum_{\substack{s(e)\in S\\ t(e)\in T}}\mathit{flow}(e) - \sum_{\substack{t(e)\in S\\ s(e)\in T}}\mathit{flow}(e)\\
&\le \sum_{\substack{s(e)\in S\\ t(e)\in T}}\mathit{flow}(e)\\
&\le \sum_{\substack{s(e)\in S\\ t(e)\in T}}\mathit{cap}(e)\\
&= \mathit{cap}(S,T).
\end{aligned}$$

9. Aus dem gerade Gezeigten folgt insbesondere, dass die in Punkt 3 bereits unterstellte Gleichheit von $\mathit{outflow}(A) - \mathit{inflow}(A)$ und $\mathit{inflow}(B) - \mathit{outflow}(B)$ tatsächlich gilt. Denn offensichtlich bilden die Mengen $V - \{B\}$ und $\{B\}$ einen Schnitt, so dass gilt:

$$\begin{aligned}
\mathit{outflow}(A) - \mathit{inflow}(A) &= \mathit{val}(\mathit{flow})\\
&= \mathit{flow}(V - \{B\}, \{B\})\\
&= \sum_{\substack{s(e)\neq B\\ t(e)=B}}\mathit{flow}(e) - \sum_{\substack{s(e)=B\\ t(e)\neq B}}\mathit{flow}(e)\\
&= \mathit{inflow}(B) - \mathit{outflow}(B).
\end{aligned}$$

10. Bezeichnet man den maximal möglichen Durchsatz mit *maxflow* und die kleinstmögliche Kapazität eines Schnittes mit *mincut*, dann folgt aus der in Punkt 8 gezeigten Beziehung:

$$\mathit{maxflow} \le \mathit{mincut}.$$

11. Nun lässt sich auch der Beweis der Behauptung in Punkt 7 vervollständigen. Sei *flow* ein Fluss, zu dem es keine aufsteigenden Pfade mehr gibt, wie ihn der Ford-Fulkerson-Algorithmus liefert. Dann kann ein Schnitt (S_0, T_0) konstruiert werden, bei dem Fluss

und Kapazität identisch sind. Mit den Punkten 8 und 10 ergibt sich daraus folgende Vergleichskette:

$$\mathit{cap}(S_0, T_0) = \mathit{flow}(S_0, T_0) \leq \mathit{maxflow} \leq \mathit{mincut} \leq \mathit{cap}(S_0, T_0),$$

was, wie gewünscht, $\mathit{flow}(S_0, T_0) = \mathit{maxflow}$ $(= \mathit{mincut})$ impliziert.

Es bleibt, (S_0, T_0) zu finden. Wähle S_0 als Menge aller Knoten, die von A aus durch einen aufsteigenden Pfad erreichbar sind (einschließlich A), und $T_0 = V - S_0$. B muss zu T_0 gehören, weil es sonst einen aufsteigenden Pfad von A nach B gäbe in Widerspruch zur Wahl von flow. Also ist (S_0, T_0) ein Schnitt. Betrachte nun eine Kante e mit $s(e) \in S_0$ und $t(e) \in T_0$. Dann gibt es einen aufsteigenden Pfad $\hat{e}_1 \cdots \hat{e}_n$ von A nach $s(e)$. Wäre $\mathit{flow}(e) < \mathit{cap}(e)$, wäre auch $\hat{e}_1 \cdots \hat{e}_n e$ ein aufsteigender Pfad, und $t(e)$ gehörte zu S_0 im Widerspruch zur Wahl von e. Also gilt $\mathit{flow}(e) = \mathit{cap}(e)$. Betrachte analog eine Kante e mit $s(e) \in T_0$ und $t(e) \in S_0$. Dann gibt es einen aufsteigenden Pfad $\hat{e}_1 \cdots \hat{e}_n$ von A nach $t(e)$. Wäre $\mathit{flow}(e) > 0$, wäre auch $\hat{e}_1 \cdots \hat{e}_n \bar{e}$ ein aufsteigenden Pfad von A nach $s(e)$, so dass $s(e)$ zu S_0 gehörte im Widerspruch zur Wahl von e. Also gilt $\mathit{flow}(e) = 0$. Zusammengefasst ergibt sich für den Fluss des Schnittes (S_0, T_0) die gewünschte Übereinstimmung mit der Kapazität:

$$\begin{aligned}
\mathit{flow}(S_0, T_0) &= \sum_{\substack{s(e) \in S_0 \\ t(e) \in T_0}} \mathit{flow}(e) - \sum_{\substack{t(e) \in S_0 \\ s(e) \in T_0}} \mathit{flow}(e) \\
&= \sum_{\substack{s(e) \in S_0 \\ t(e) \in T_0}} \mathit{cap}(e) - \sum_{\substack{t(e) \in S_0 \\ s(e) \in T_0}} 0 \\
&= \mathit{cap}(S_0, T_0).
\end{aligned}$$

Kapitel 7

Ein merkwürdiger Dialog über merkwürdige Graphprobleme

Der folgende Dialog zwischen P. Enpe, Chef eines Softwarehauses, und seinem Mitarbeiter Epimetheus ist einem ähnlichen Dialog in dem Buch von Garey und Johnson [GJ79] nachempfunden.

P: Die Graphprobleme der kürzesten Wege und minimalen aufspannenden Bäume –

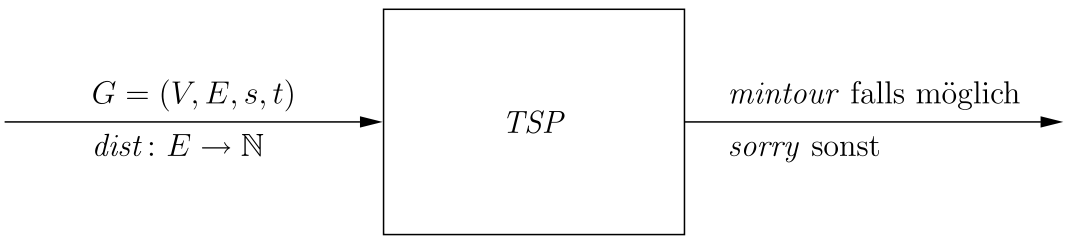
1. *shortest-path* mit einmaligem Besuch aller Knoten (Travelling-Salesman-Problem)
2. *shortest-path* mit einer Auswahl von Zwischenstationen
3. *shortest-path* mit verschiedenen Kunden und verschiedenen Lagern an verschiedenen Orten sowie mit der Beschränkung von Transportkapazitäten (Tourenplanungsproblem)
4. *min-span-tree* mit beschränkter Verzweigung
5. *min-span-tree* mit Mindesthöhe
6. Berechnung der minimalen Färbung

– und Ähnliches mehr sollen profitabel realisiert werden.

E: Jawohl, Chef!

(4 Wochen später)

E: Here we are:



Dabei generiert *TSP* alle einfachen Wege der Länge $\#V - 1$ und bestimmt unter diesen den Weg mit der kürzesten Entfernung.

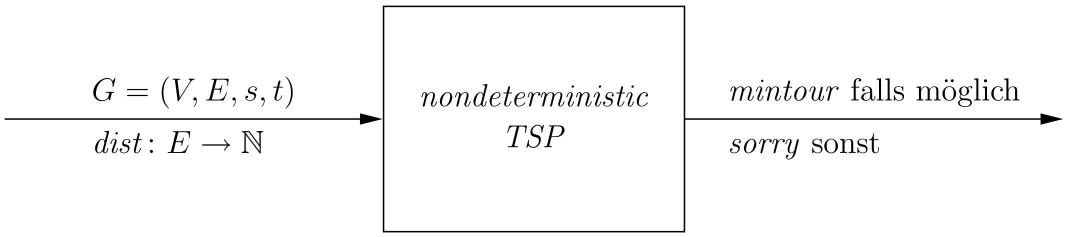
P: Phantastisch. Und die Qualität?

E: Korrekt und benutzungsfreundlich, doch langsam. Denn die Zahl der einfachen Wege der Länge $\#V - 1$ kann $\#V!$ erreichen, d.h. exponentiell sein.

P: Das muss anders werden.

(6 Wochen später)

E: Jetzt sind wir dem Ziel nahe:



Dabei wählt *nonderministic TSP* nach bestmöglichem Startknoten jeweils bestmöglichen Folgeknoten. Auch die Qualität stimmt: korrekt, benutzungsfreundlich, schnell. ...

P: Aber?

E: Es läuft nicht, weil ein Systemteil fehlt, ein sogenanntes „Orakel“, das den bestmöglichen Knoten zu wählen erlaubt.

P: Kaufen!

E: Haben wir versucht: eine Firma liefert erst im Herbst. Die einzige andere am Markt kann sofort liefern. Doch schon bei der Vorführung haben einige Auswahlfälle eine exponentielle Laufzeit gezeigt.

P: Bauen!

E: Haben wir versucht mit *Tiefensuche*, *Breitensuche*, *Bestensuche* u.ä. Ist in ungünstigen Fällen alles exponentiell.

P: Ich will Erfolge sehen.

(8 Wochen später)

E: Es geht vielleicht gar nicht.

P: Vielleicht?

E: Wir haben versucht zu zeigen, dass es nicht geht, sind aber hoffnungslos gescheitert.

P: Stümper!

E: Es scheint schwierig.

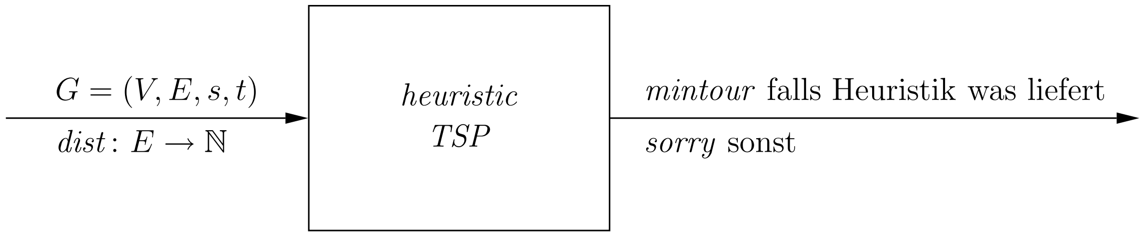
P: Scheint?

E: Wir haben 77 Probleme gefunden, die alle schnell lösbar sind, wenn unsere Probleme schnell lösbar sind. Und unsere Probleme sind schnell lösbar, wenn es eines der 77 ist. Außerdem geht es Hunderten von Forschungsgruppen und Entwicklungsteams wie uns.

P: Ich will Erfolge sehen.

(12 Wochen später)

E: Wir haben es:



Der Algorithmus *heuristic TSP* wählt Anfangsknoten und jeweils nächsten Knoten nach einem schnellen Kriterium (*Heuristik*) aus.

Und zur Qualität lässt sich sagen: schnell und benutzungsfreundlich, doch inkorrekt.

P: Ich wusste, dass Sie es schaffen.

Kapitel 8

NP-vollständige Graphprobleme

Die Graphprobleme der kürzesten Wege, minimalen aufspannenden Bäume und maximalen Flüsse sowie ähnlich gelagerte haben algorithmische Lösungen mit polynomiellem Zeitaufwand, liegen also im Bereich des Machbaren, sind einen Versuch wert.

Bei vielen anderen Graphproblemen ist die algorithmische Sachlage wesentlich komplizierter. Für sie sind Lösungen mit exponentieller Laufzeit bekannt, außerdem können nichtdeterministische Lösungen, bei denen in jedem Schritt die beste aus einer beschränkten Anzahl von Möglichkeiten geraten werden darf, mit polynomieller Laufzeit angegeben werden. Dafür einige Beispiele.

8.1 Beispiele

1. **Traveling-Salesman-Problem (TSP)**

   Gibt es einen Kreis in einem ungerichteten Graphen, der jeden Knoten genau einmal besucht und dessen Entfernung kleiner oder gleich einer vorgegebenen Zahl ist?

   Eingabe: ungerichteter Graph $G = (V, E)$ mit $V = \{1, \ldots, n\}$,
   Entfernung $dist\colon E \to \mathbb{N}$ sowie
   $N \in \mathbb{N}$.

   Ausgabe: JA, wenn eine Permutation $i_1 \cdots i_n$ existiert mit $e_j = \{i_j, i_{j+1}\} \in E$ für $j = 1, \ldots, n-1$, $e_n = \{i_n, i_1\} \in E$ und $\sum_{j=1}^{n} dist(e_j) \leq N$;
   NEIN sonst.

   deterministisches Verfahren: Probiere alle Permutationen.

   Aufwand: (im schlechtesten Fall mindestens) $n! \geq 2^{n-1}$.

   nichtdeterministisches Verfahren: Beginne bei beliebigem i_1 und rate für $j = 2, \ldots, n$ nächstbeste Station i_j.

   Aufwand: n (Auswahl pro Schritt durch n beziehungsweise maximalen Knotengrad beschränkt).

Beachte, dass man eine explizite Rundreise erhält, wenn man sich die Permutation merkt, die JA liefert. Will man eine kleinste Rundreise, beginnt man mit einem N, das JA ergibt, und erniedrigt es Schritt für Schritt um 1, bis NEIN kommt. Wählt man als N das n-fache der maximalen Entfernung einer Kante, so ist die Antwort JA, wenn es überhaupt eine Rundreise gibt.

2. **Hamiltonkreis-Problem (HAM)**

   Gibt es einen Kreis, auf dem jeder Knoten genau einmal vorkommt?

   Eingabe: ungerichteter Graph $G = (V, E)$ mit $V = \{1, \ldots, n\}$.

   Ausgabe: JA, wenn eine Permutation $i_1 \cdots i_n$ mit $\{i_1, i_2\}, \ldots, \{i_{n-1}, i_n\}, \{i_n, i_1\} \in E$ existiert;
   NEIN sonst.

   deterministisches und nichtdeterministisches Verfahren: wie **TSP**

3. **Färbungsproblem (COLOR)**

   Gibt es für einen ungerichteten Graphen eine Färbung mit k Farben?

   Eingabe: ungerichteter Graph $G = (E, V)$ und
   $k \in \mathbb{N}$.

   Ausgabe: JA, wenn es eine Abbildung $color\colon V \to [k]$ gibt mit $[k] = \{1, \ldots, k\}$ und $color(v) \neq color(v')$ für alle $e = \{v, v'\} \in E$;
   NEIN sonst.

   deterministisches Verfahren: Probiere alle Abbildung $color\colon V \to [k]$.

   Aufwand: (im schlechtesten Fall mindestens) Zahl der Abbildungen k^n mit $n = \#V$.

   nichtdeterministisches Verfahren: Rate für jeden Knoten die richtige Farbe.

   Aufwand: Zahl der Knoten $n = \#V$.

   Beachte, dass man die Färbungszahl erhält, wenn man mit $k = 1$ beginnt und dann solange um 1 erhöht, bis die Antwort JA kommt.

4. **Cliquenproblem (CLIQUE)**

   Gibt es in einem ungerichteten Graphen einen vollständigen Teilgraph mit m Knoten?

   Eingabe: ungerichteter Graph $G = (V, E)$ und
   $m \in \mathbb{N}$.

   Ausgabe: JA, wenn $X \subseteq V$ existiert mit $\#X = m$ und $\binom{X}{2} \subseteq E$, d.h. $\{v, v'\} \in E$ für alle $v, v' \in X$ mit $v \neq v'$;
   NEIN sonst.

   deterministisches Verfahren: Probiere für alle m-elementigen Teilmengen $V' \subseteq V$, ob $\{v, v'\} \in E$ für alle $v, v' \in V'$ mit $v \neq v'$.

Aufwand: Zahl der m-elementigen Teilmengen $\binom{n}{m}$ (was für m in der Nähe von $\frac{n}{2}$ exponentiell ist).

nichtdeterministisches Verfahren: Rate die Elemente von X.

Aufwand: höchstens n.

Wie bekommt man die größte Clique oder zumindest ihre Knotenzahl?

In diesen und vielen anderen Beispielen ist die Situation ähnlich: Bekannte ad-hoc-Lösungen, bei denen im Prinzip alle Möglichkeiten durchprobiert werden, sind exponentiell; ob es aber polynomielle Lösungen gibt, ist unbekannt. Stattdessen findet man aber meist relativ leicht nichtdeterministische Verfahren, bei denen jede Berechnung polynomiell lange läuft, pro Rechenschritt aber eine Auswahl vorliegen kann und die positive Antwort nur richtig gefunden wird, wenn man die richtige Wahl trifft. Wenn für eine Eingabe die Antwort JA ist, muss nur eine Berechnung diese liefern, alle anderen Berechnungen dürfen zu einem falschen Ergebnis führen; wenn die Antwort NEIN ist, müssen alle Berechnungen dies bestätigen, darf also keine JA ergeben. Leider ist es bisher nicht gelungen, Programme zu entwickeln oder Maschinen zu bauen, die das richtige Raten auch wirklich in polynomieller Zeit schaffen.

8.2 Reduktion und NP-Vollständigkeit

Nach den obigen Überlegungen gehören alle betrachteten Beispiele zur Klasse NP der Probleme, die sich durch einen nichtdeterministischen Algorithmus mit polynomiellem Aufwand lösen lassen. Dabei beschränkt man sich auf Entscheidungsprobleme der Form $D\colon IN \to \{\mathrm{JA}, \mathrm{NEIN}\}$, die jeweils einen Eingabebereich IN besitzen und für jede Eingabe entweder JA oder NEIN liefern.

Wenn P die Klasse der Probleme bezeichnet, die sich durch einen deterministischen Algorithmus in polynomieller Zeit lösen lassen, dann gilt offenbar $\mathrm{P} \subseteq \mathrm{NP}$ (denn „nichtdeterministisch“ bedeutet, dass in jedem Rechenschritt mehrere Möglichkeiten zur Auswahl stehen können).

Es ist dagegen völlig unklar, ob $\mathrm{NP} \subseteq \mathrm{P}$ gilt oder nicht. Die positive Antwort wäre von immenser praktischer Bedeutung, weil viele wichtige Probleme in NP liegen. Die negative Antwort wird allerdings von den meisten Fachleuten erwartet und hätte zumindest im theoretischen Bereich nachhaltige Konsequenzen. Wer die Antwort findet, kann jedenfalls berühmt werden.

Bei den vielfältigen Klärungsversuchen hat sich eine Klasse von Problemen in NP als besonders signifikant erwiesen, nämlich die NP-vollständigen. Denn ein Einziges davon könnte das Rätsel entschlüsseln. Um das einzusehen, braucht man das Konzept der Reduktion, das auch sonst sehr nützlich ist.

Damit $\mathrm{P} = \mathrm{NP}$ gilt, muss man für jedes Problem in NP eine polynomielle Lösung finden. Allerdings ist unklar, wie das gehen soll. In solchen Fällen versucht man gern, die Gesamtaufgabe zu zerlegen. Gelingt es beispielsweise einige NP-Probleme in polynomieller Zeit auf andere zurückzuführen (zu reduzieren), braucht man „nur“ noch für die schwierigeren Probleme zu zeigen, dass sie in P liegen, denn die Hintereinanderschaltung polynomieller Algorithmen ist polynomiell.

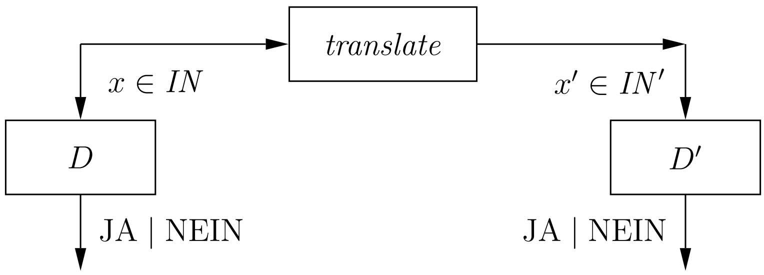
Vor allem wäre ein NP-Problem interessant, auf das sich alle andereren reduzieren lassen, weil dann NP $\subseteq$ P ist, sobald dieses eine in P liegt. Solche Probleme werden NP-vollständig genannt.

1. **Reduktion**

   Seien D und D' Entscheidungsprobleme mit den Eingabemengen IN bzw. IN' und den Ausgaben JA und NEIN. Dann heißt D auf D' *reduzierbar* (in Zeichen: $D \leq D'$), falls es eine berechenbare Übersetzungsfunktion $\mathit{translate}\colon IN \to IN'$ mit polynomiell beschränkter Berechnungslänge gibt, so dass gilt:

   $$D(x) = \text{JA} \quad \text{gdw.} \quad D'(\mathit{translate}(x)) = \text{JA}.$$

   Die Situation, die mit dem Reduktionsbegriff verbunden ist, lässt sich durch folgendes Diagramm veranschaulichen.

   Um also festzustellen, dass $D \leq D'$ gilt, muss man eine berechenbare Übersetzung der Eingaben von D in die Eingaben von D' konstruieren, die polynomiell ist sowie die Antwort JA bzgl. D in die Antwort JA bzgl. D' überträgt und umgekehrt. Diese Eigenschaft bzgl. der Ausgabe JA nennt man Korrektheit. Die Korrektheit bzgl. JA impliziert die Korrektheit bzgl. NEIN, weil es nur zwei Ausgaben gibt, und wenn die eine immer gleichzeitig auftritt, muss auch die andere immer gleichzeitig auftreten. Statt $D'(\mathit{translate}(x)) = \text{JA}$ impliziert $D(x) = \text{JA}$ könnte man aus demselben Grund auch $D(x) = \text{NEIN}$ impliziert $D'(\mathit{translate}(x)) = \text{NEIN}$ fordern.

   Beobachtung

   $D \leq D'$ und $D' \in$ P impliziert $D \in$ P.

   Beweis: D ist nach Voraussetzung die sequentielle Komposition von einer polynomiellen Übersetzung und dem polynomiellen D' und somit selbst polynomiell.

Die Bedeutung der NP-Vollständigkeit ist aus folgendem Ergebnis ersichtlich, das auch eine Möglichkeit nennt, wie NP-Vollständigkeit nachgewiesen werden kann.

2. **NP-Vollständigkeit**

   $D_0 \in$ NP ist NP-*vollständig*, wenn $D \leq D_0$ für alle $D \in$ NP.

   Beobachtung

   Sei D_0 NP-vollständig. Dann gilt:

(1) $D_0 \in \mathrm{P}$ gdw. $\mathrm{P} = \mathrm{NP}$.

(2) $D_0 \leq D_1$ und $D_1 \in \mathrm{NP}$ impliziert D_1 NP-vollständig.

Beweis:

(1) $\mathrm{P} = \mathrm{NP}$ impliziert $D_0 \in \mathrm{P}$, denn nach Voraussetzung ist $D_0 \in \mathrm{NP}$.
Sei umgekehrt $D_0 \in \mathrm{P}$. Dann ist $\mathrm{NP} \subseteq \mathrm{P}$ zu zeigen (da $\mathrm{P} \subseteq \mathrm{NP}$ klar ist). Sei dazu $D \in \mathrm{NP}$. Dann gilt $D \leq D_0$, so dass D die sequentielle Komposition einer polynomiellen Übersetzung und des polynomiellen D_0 ist und somit selbst polynomiell.

(2) $D_0 \leq D_1$ impliziert eine polynomielle Übersetzung $\mathit{translate}'\colon IN_0 \to IN_1$, so dass gilt:

$$D_0(y) = \text{JA} \ \text{ gdw. } \ D_1(\mathit{translate}'(y)) = \text{JA}. \tag{$*$}$$

Sei nun $D \in \mathrm{NP}$. Dann ist $D \leq D_0$, da D_0 NP-vollständig ist. Also existiert eine polynomielle Übersetzung $\mathit{translate}\colon IN \to IN_0$, so dass gilt:

$$D(x) = \text{JA} \ \text{ gdw. } \ D_0(\mathit{translate}(x)) = \text{JA}. \tag{$**$}$$

Die Komposition $\mathit{translate}' \circ \mathit{translate}\colon IN \to IN'$ ist dann eine polynomielle Übersetzung, so dass wegen $(*)$ und $(**)$ gilt:

$$D(x) = \text{JA} \ \text{ gdw. } \ D_0(\mathit{translate}(x)) = \text{JA} \ \text{ gdw. } \ D_1(\mathit{translate}'(\mathit{translate}(x))) = \text{JA}.$$

Mit anderen Worten gilt: $D \leq D_1$, und da D beliebig gewählt ist, erweist sich D_1 als NP-vollständig.

Bevor allerdings dieses Ergebnis zum Tragen kommen kann, braucht man überhaupt NP-vollständige Probleme. Für ein Einzelnes davon kann man dann zu zeigen versuchen, dass es in P oder $\mathrm{NP} - \mathrm{P}$ liegt. Und mit Punkt 2 lassen sich weitere Kandidaten für diese Versuche finden. Als eines der bemerkenswertesten Ergebnisse der Theoretischen Informatik wurde 1971 von Cook nachgewiesen, dass das Erfüllbarkeitsproblem NP-vollständig ist.

8.3 Erfüllbarkeitsproblem

Aussagen der Aussagenlogik können gelten oder nicht. Die Feststellung des einen oder anderen wird Erfüllbarkeitsproblem genannt. Um besser damit arbeiten zu können, werden die Aussagen in konjunktiver Normalform betrachtet:

(1) Es gibt Grundaussagen (Boolesche Variablen) $x \in X$, die den Wert 1 (wahr) oder 0 (falsch) annehmen können;

(2) Literale sind Grundaussagen oder negierte Grundaussagen $\bar{x}$ für $x \in X$; $\bar{x}$ hat den Wert 1, wenn x den Wert 0 hat, und den Wert 0, wenn x den Wert 1 hat;

(3) Klauseln sind Disjunktionen $c = l_1 \vee l_2 \vee \cdots \vee l_m$ von Literalen l_i; c hat den Wert 1, wenn mindestens ein l_i den Wert 1 hat, sonst 0;

(4) Formeln sind Konjunktionen $f = c_1 \wedge c_2 \wedge \cdots \wedge c_n$ von Klauseln c_j; f hat den Wert 1, wenn alle c_j den Wert 1 haben, und sonst den Wert 0.

Problem (SAT): Gibt es eine Belegung $ass\colon X \to \{0,1\}$, so dass eine Formel f den Wert 1 hat (d.h. gilt bzw. erfüllt ist)?

Lösung: (1) Probiere alle Belegungen.

(2) Rate die richtige für jedes $x \in X$.

Aufwand: (1) Zahl der Abbildungen von X nach $\{0,1\}$, also 2^k, wenn $k = \#X$.

(2) k-mal Raten, d.h. **SAT** $\in$ NP.

Theorem
SAT ist NP-vollständig.

Auf den schwierigen Beweis wird verzichtet, weil das Ergebnis nicht zum Gebiet der Algorithmen auf Graphen gehört, sondern hier nur benutzt wird.

Für manche Zwecke günstiger ist das Problem **SAT3**, das ist das Erfüllbarkeitsproblem, wobei allerdings nur Formeln eingegeben werden, deren Klauseln aus jeweils drei Literalen bestehen.

Korollar:
SAT $\leq$ **SAT3**, so dass **SAT3** NP-vollständig ist.

8.4 NP-vollständige Graphenprobleme

Das Erfüllbarkeitsproblem erlaubt den Einstieg in die Klasse der NP-vollständigen Graphenprobleme, denn es gilt:

SAT3 $\leq$ **CLIQUE**, so dass auch **CLIQUE** NP-vollständig ist.

Durch ähnliche Überlegungen stellt sich heraus, dass **HAM**, **TSP** u.v.a.m. NP-vollständig sind.

Beweis für **SAT3** $\leq$ **CLIQUE**: Dazu ist zuerst ein polynomieller Übersetzer nötig, der eine Formel $f = c_1 \wedge \cdots \wedge c_n$ mit $c_i = l_{i1} \vee l_{i2} \vee l_{i3}$ in einen geeigneten Graphen und eine Zahl umwandelt:

$$G(f) = ([n] \times [3], E(f))$$

mit $E(f) = \{\{(i,j),(i',j')\} \mid i \neq i',\ l_{ij} \text{ und } l_{i'j'} \text{ ohne Widerspruch}\}$.

Zu jedem Literal der Formel gibt es also einen Knoten, und Literale aus verschiedenen Klauseln werden verbunden, wenn sie sich nicht widersprechen, wobei sich zwei Literale nur dann widersprechen, wenn das eine die Negation des anderen ist. Als gesuchte Cliquengröße wird n gewählt. (Beachte, dass keine Clique größer sein kann, weil die Literalknoten aus derselben Klausel nicht durch Kanten verbunden sind.)

Es bleibt zu zeigen, dass f genau dann erfüllbar ist, wenn $G(f)$ eine n-Clique besitzt.

Sei f erfüllbar, d.h. es gibt Literale $l_{ij(i)}$ für $i = 1, \ldots, n$ (also je eins aus jeder Klausel), die alle gleichzeitig gelten können. Keine zwei bilden also einen Widerspruch, so dass die Literale als Knoten paarweise verbunden sind und somit eine n-Clique bilden.

Sei umgekehrt $\{v_1, \ldots, v_n\}$ eine Knotenmenge aus $G(f)$, die eine Clique bildet, d.h. für $k \neq l$ ist $\{v_k, v_l\} \in E(f)$. Für jeden Knoten v_k gibt es Indizes $i(k), j(k)$ mit $v_k = (i(k), j(k))$ und die Literale $l_{i(k)j(k)}$ und $l_{i(l)j(l)}$ widersprechen sich nicht, weil sie als Knoten durch eine Kante verbunden sind. Also können die Literale $l_{i(k)j(k)}$ für $k = 1, \ldots, n$ alle gleichzeitig gelten. Außerdem kommen die Literale je aus verschiedenen Klauseln. Denn wäre $i(k) = i(l)$ für $k \neq l$, dann ist $j(k) \neq j(l)$ (sonst wäre $v_k = v_l$), und man hätte eine Kante zwischen zwei Literalen aus derselben Klausel, was nach Konstruktion von $G(f)$ nicht geht. Literale aus allen Klauseln, die gleichzeitig gelten, erfüllen aber gerade f.

Insgesamt hat man damit eine Reduktion von **SAT3** auf **CLIQUE**.

Kapitel 9

Was tun, solange das Orakel fehlt?

NP-Probleme und insbesondere NP-vollständige Probleme sind vielfach von großer praktischer Bedeutung, so dass sie algorithmisch bearbeitet werden müssen, auch wenn sie tatsächlich exponentiell sind. Aber selbst wenn sie sich doch eines Tages wider Erwarten als polynomiell lösbar erwiesen, muss jetzt nach Auswegen gesucht werden, solange exakte Lösungen nicht effizient implementiert sind.

Im folgenden werden einige Möglichkeiten vorgestellt.

9.1 Lokale Suche und Optimierung

Bei Optimierungsproblemen (und analog bei Fragen, ob sich gegebene Schranken über- oder unterschreiten lassen) gibt es immer eine Lösungsmenge, oft *Lösungsraum* genannt, die bezüglich der zu optimierenden Größe geordnet ist.

Wenn es gelingt, Operationen zu finden, die aus einer gegebenen Lösung in polynomieller Zeit eine bessere machen, kann durch wiederholte Anwendung der Operationen von einer Anfangslösung aus eine Lösung hergestellt werden, die sich durch die Operationen nicht mehr verbessern lässt. Wenn nur polynomiell viele Schritte hintereinander möglich sind, ist die optimierte Lösung in polynomieller Zeit erreicht. Im allgemeinen wird man allerdings nur ein lokales Optimum haben. Und wie weit es vom globalen Optimum entfernt ist, weiß man meist auch nicht.

Ein Beispiel für dieses Vorgehen sind die beiden Operationen *Zusammenlegen von Touren* und *Tauschen eines Auftrages von einer Tour zu einer anderen* bei der Tourenplanung.

Bei der *lokalen Suche* wird eine schnelle Lösung erreicht, weil man auf die eigentlich gewünschte Optimalität der Resultate verzichtet und sich mit einem viel schwächeren Kriterium zufrieden gibt.

9.2 Abschwächung der Lösungsanforderungen

Einen ähnlichen Effekt kann man erreichen, wenn man andere Forderungen an die Lösungen abschwächt, also praktisch den Lösungsraum erweitert. Wenn man Glück oder Geschick hat, lässt sich das Optimum im neuen Lösungsraum schnell finden. Ist das Resultat auch eine Lösung im ursprünglichen Sinne, hat man das Optimum auch für das ursprüngliche Problem berechnet. Doch im allgemeinen wird das Resultat nur eine Lösung im abgeschwächten Sinne sein. Aber selbst wenn diese für die eigentliche Aufgabe nicht verwendet wird, liefert sie wichtige Erkenntnisse. Das Optimum im erweiterten Lösungsraum ist nämlich eine untere Schranke für das eigentlich gesuchte Optimum. Man bekommt also eine Ahnung, wie weit man mit einer Lösung noch vom Optimum entfernt sein kann.

Lässt man beispielsweise bei der Lösung eines Travelling-Salesman-Problems eine Kante weg, entsteht ein – ziemlich degenerierter – aufspannender Baum. Der Wert eines minimalen aufspannenden Baumes, der bekanntlich polynomiell bestimmt werden kann, bildet somit eine untere Schranke für die Entfernung von Rundreisen.

9.3 Untere Schranken

Auch wenn man die Anforderungen an Lösungen nicht abschwächt, können untere Schranken helfen. Denn eine Lösung, die man auf welchem Wege auch immer erhalten hat, muss relativ gut sein, wenn sie in der Nähe einer unteren Schranke liegt. Untere Schranken lassen sich auch vorteilhaft in der Branch&Bound-Methode einsetzen.

Untere Schranken lassen sich nicht nur durch Abschwächen der Lösungsanforderungen erreichen wie in Punkt 2, sondern auch durch andere Überlegungen.

Für das Travelling-Salesman-Problem zum Beispiel erhält man für jeden gerichteten Graphen $M = (V, E, s, t)$ und jede Entfernungsfunktion $dist\colon E \to \mathbb{N}$ mit folgender Beobachtung eine untere Schranke: Eine Rundreise besucht jeden Knoten einmal, indem sie mit einer Kante ankommt und mit einer weggeht (wobei jede Kante einmal ankommt und einmal weggeht). Summiert man also für jeden Knoten die kleinsten Entfernungen eingehender und ausgehender Kanten auf, kann keine Rundreise eine kleinere Entfernung als die Hälfte dieser Summe haben:

$$bound(M, dist) = \tfrac{1}{2} \cdot \sum_{v \in V} \big(\min_{e \in E}\{ dist(e) \mid s(e) = v\} + \min_{e \in E}\{ dist(e) \mid t(e) = v\} \big).$$

Die Summe der Minima der eingehenden Kanten bzw. die entsprechende Summe der ausgehenden Kanten bilden ebenfalls untere Schranken, zwischen denen $bound(M, dist)$ liegt. Am besten nimmt man das Maximum der beiden Summen.

9.4 Spezielle Graphen

Ein (NP-vollständiges) Problem auf Graphen kann in polynomieller Zeit lösbar werden, wenn nicht mehr alle Graphen als Eingabe zugelassen werden, sondern spezielle, die dann vielleicht aber nicht mehr praktisch interessant sind.

Zur Illustration sei das Cliquen-Problem betrachtet. Sei $b \in \mathbb{N}$ und G ein Graph, dessen Knotengrad, d.h. die maximale Zahl von Kanten an einem Knoten, b nicht überschreitet. Dann hat die größte Clique von G höchstens $b+1$ Elemente. Die Frage nach noch größeren Cliquen kann also immer sofort verneint werden. Die Frage nach einer Clique der Größe $k \leq b+1$ erfordert höchstens die Überprüfung aller k-elementigen Teilmengen der Knotenmenge. Davon gibt es $\binom{n}{k}$, wenn n die Zahl der Knoten ist, und somit nicht mehr als n^{b+1}. Und zu überprüfen, ob k Knoten eine Clique bilden, geht sowieso schnell.

Ähnliches ergibt sich für planare Graphen, von denen man weiß, dass sie keine Clique mit 5 Knoten enthalten können. Nur die Frage nach Cliquen mit 3 oder 4 Knoten hat also keine Standardantwort. Und für 3 und 4 geht es schnell, wie gerade für $k \leq b+1$ allgemeiner gezeigt wurde.

Literaturverzeichnis

[AH77] K. Appel and W. Haken. Every planer map is 4-colorable. *Illinois J. Math.*, 21:429–567, 1977.

[Aig84] M. Aigner. *Graphentheorie – Eine Entwicklung aus dem 4-Farben-Problem.* Teubner, Stuttgart, 1984.

[Coo71] S.A. Cook. The complexity of theorem-proving procedures. In *Proc. STOC '71*, pages 151–158. ACM, New York, 1971.

[Eve79] S. Even. *Graph Algorithms.* Computer Science Press, Rockville, 1979.

[Gib85] A. Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, Cambridge, New York, Port Chester, Melbourne, Sydney, 1985.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and intractability – A guide to the theory of NP-completeness.* W.A. Freeman and Company, New York, 1979.

[Jun90] D. Jungnickel. *Graphen, Netzwerke und Algorithmen.* BI Wissenschaftsverlag, Mannheim, Wien, Zürich, 1990.

[Meh84] K. Mehlhorn. *Graph Algorithms and NP-Completeness.* Springer, Berlin, Heidelberg, New York, Tokyo, 1984.