Algorithmen auf Graphen

Hans-Jörg Kreowski[1]
Universität Bremen
Studiengang Informatik
kreo@informatik.uni-bremen.de

10. Dezember 2002

[1]Kapitel 4 entspricht nicht der aktuellen Behandlung des Themas in der Lehrveranstaltung. Außerdem fehlen noch zwei Illustrationen.

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

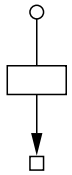
Wohlstrukturierte Flussdiagramme sind graphische Darstellungen der Anweisungen einer einfachen, PASCAL–ähnlichen Programmiersprache. Die Syntax sieht Wertzuweisung, Reihung, Fallunterscheidung *(if–then–else)* und Wiederholung *(while–do)* vor; die zugehörige Backus–Naur–Form lautet:

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

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:

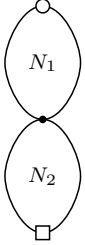
(a) Sie haben immer genau einen Eingang und einen Ausgang, die Knoten sind und mit *begin* bzw. *end* bezeichnet werden.

(b) Eine Wertzuweisung wird als Kante dargestellt:

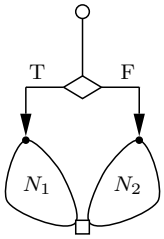


Der Kasten in der Mitte der Kante kann als Markierung aufgefasst werden. Er dient lediglich der Verstärkung des optischen Eindrucks, damit die beschriebenen Graphen, auch wie Flussdiagramme aussehen und diese nicht nur formal darstellen. Man beachte, dass nur die Tatsache repräsentiert ist, dass eine 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 $N1$ und $N2$ aufteilen lässt, die beide Flussdiagramme darstellen, derart dass der Eingang von $N1$ der Eingang von N, der Ausgang von $N2$ der Ausgang von N und der Ausgang von $N1$ und der Eingang von $N2$ der einzige gemeinsame Knoten von $N1$ und $N2$ in N ist:

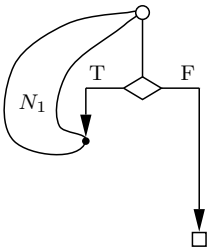


(d) Ein Graph N ist ebenfalls ein Flussdiagramm, das der Fallunterscheidung entspricht, wenn er sich folgendermaßen aufteilen lässt:



Dabei müssen $N1$ und $N2$ Flussdiagramme sein, deren Ausgang auch Ausgang von N ist und die sonst keine gemeinsamen Knoten haben. Außerdem muss 1 der Eingang von $N1$ sein und 2 der von $N2$. Neben seinem Eingang muss N noch einen weiteren Knoten besitzen, der mit der Rombusform 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:

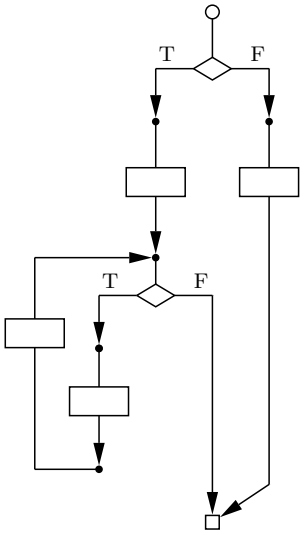


Dabei muss $N1$ ein Flussdiagramm sein, dessen Eingang der Knoten 0 und dessen Ausgang der Eingang von N ist. Außerdem besitzt N neben seinem Ausgang noch einen Knoten mit drei anhängenden Kanten wie bei den Fallunterscheidung, wobei hier allerdings die mit F markierte Kante zum Ausgang geht und die 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:

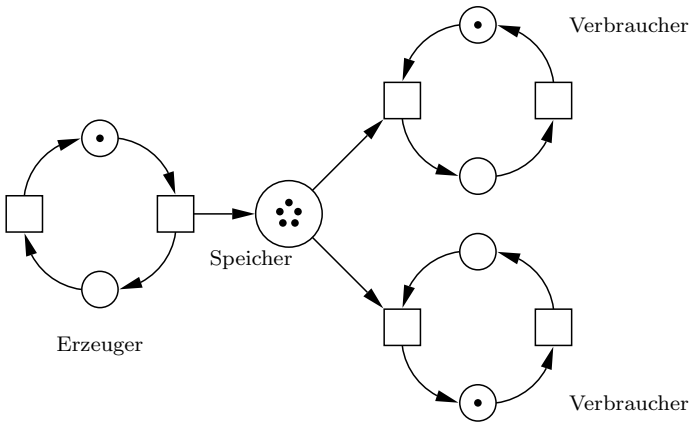
```
if   n > 0
then exp := 1;
     while n > 0 do n := n - 1;
                    exp := exp * x
else exp := 0
```

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

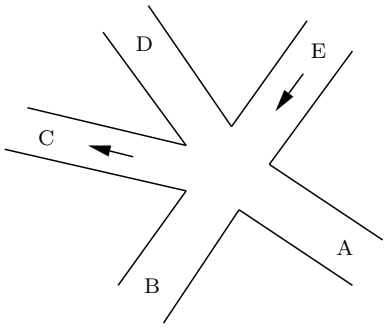
Die als Kreis gezeichneten Knoten stellen Systemeigenschaften oder -zustände dar. Ihre Markierung ist immer eine natürliche Zahl, die Zahl der im Kreis befindlichen Punkte. Ihr Wert sagt etwas über die konkrete Beschaffenheit der jeweiligen Eigenschaften. Im einzelnen beschreiben die Knoten, die mit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

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

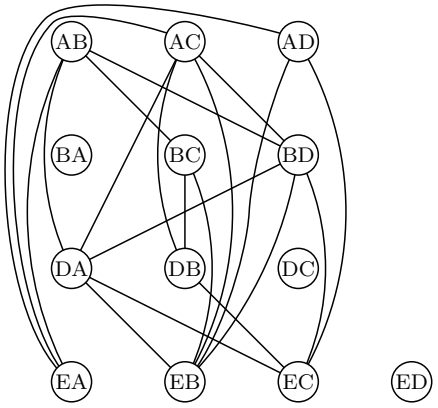
Für die Knoten 5 bis 8 bedeutet die Markierung 0, dass die genannte Eigenschaft gerade nicht vorliegt, während 1 die Erfülltheit beschreibt. Alle anderen Zahlen machen keinen Sinn. Die Markierung des Knotens 9 gibt die Zahl der momentan im Warenlager gespeicherten Warenheinheiten an.
Es ist klar, dass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

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 $f : V \to C$ *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, dass 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 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 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 3

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 werden und bei den 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 Kürzesten-Wege-Problems werden für gerichtete Graphen gemacht, weil für die der Wegebegriffe etwas einfacher ist. Außerdem können sie etwas flexibler angewendet werden, weil die Entfernung zwischen zwei Orten nicht unbedingt symmetrisch sein muss.

1. Ein *gerichteter Graph* $M = (V, E, s, t)$ besteht aus einer Menge V von *Knoten*, einer Menge E von *Kanten*, und zwei Abbildungen $s : E \to V$ und $t : E \to V$, 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 $dist : E \to \mathbb{N}$.

   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: $dist : E^* \to \mathbb{N}$ 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 *Kürzesten-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).

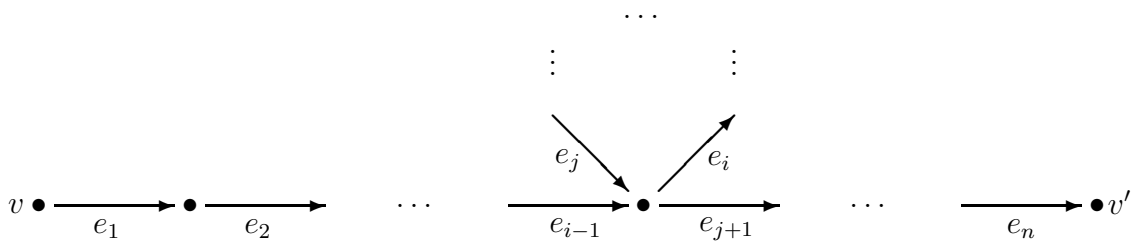
10. Ein Weg von v nach v der Länge $n \geq 1$ wird auch *Kreis* genannt.

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 $e_i \ldots e_j$ 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 $\sharp V$ die Knotenzahl des Graphen M. Sei $p = e_1 \cdots e_n$ ein Weg mit $n \geq \sharp V$. 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: $s(e_{i+1}) = t(e_i) = v_i = v_j = t(e_j)$.

15. Sei $\sharp E$ die Kantenzahl des Graphen M. Dann gibt es $(\sharp E)^n$ Kantenfolgen der Länge n und damit höchstens $(\sharp E)^n$ 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 Kürzesten-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', als 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 $\overline{v}$ und ein Weg p'' von $\overline{v}$ 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 $\overline{v} \in \overline{\overline{V}}$ ist, dann vermeidet $p' \bullet p''$ zwischendurch die Knoten aus $\overline{V} = \overline{\overline{V}} - \{\overline{v}\}$. 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 lassen, die mehr Knoten vermeiden. Insgesamt erhält man so einen kubischen Algorithmus, der das *Kürzeste-Wege-Problem* löst.

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

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

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

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

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

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

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

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

    Das ist noch relativ leicht einzusehen. Nach Voraussetzung ist $\mathit{inter}(p_0) \cap \overline{V} = \emptyset$. Betrachte zuerst den Fall, dass $\overline{v} \notin \mathit{inter}(p_0)$. Dann ist $p_0 \in \mathit{SIMPLE}(v, v', \overline{\overline{V}})$. Außerdem gilt offenbar $\mathit{SIMPLE}(v, v', \overline{\overline{V}}) \subseteq \mathit{SIMPLE}(v, v', \overline{V})$, 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 $\overline{v} \in \mathit{inter}(p_0)$, was aber gerade die Existenz eines $i_0 \in [n-1]$ bedeutet mit $t(e_{i_0}) = \overline{v}$. Man kann dann p_0 in $\overline{v}$ 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 $\overline{p}_1 \in \mathit{SHORT}(v, \overline{v}, \overline{\overline{V}})$ mit $\mathit{dist}(\overline{p}_1) < \mathit{dist}(p_1)$. Dann kann man $\overline{p}_1 \bullet p_2$ als Weg von v nach v' bilden, der $\overline{V}$ vermeidet. Außerdem gilt

$dist(\overline{p}_1 \bullet p_2) = dist(\overline{p}_1) + dist(p_2) < dist(p_1) + dist(p_2) = dist(p_1 \bullet p_2) = dist(p_0)$. Das ist ein Widerspruch zu $p_0 \in SHORT(v, v', \overline{V})$, so dass $p_1 \in SHORT(v, \overline{v}, \overline{\overline{V}})$ sein muss. Genauso zeigt man, dass $p_2 \in SHORT(\overline{v}, v', \overline{\overline{V}})$ ist.

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

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

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

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

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

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
    $short_2(v, v') = short_2(v, v', \emptyset)$,
    $short_2(v, v', \overline{V}) = min(short_2(v, v', \overline{\overline{V}}), short_2(v, \overline{v}, \overline{\overline{V}}) + short_2(\overline{v}, v', \overline{\overline{V}}))$ für $v \neq v'$,
    $short_2(v, v, \overline{V}) = 0$,
    $short_2(v, v', V) = min(\{dist(e) \mid e \in E, s(e) = v, t(e) = v'\})$ für $v \neq v'$.

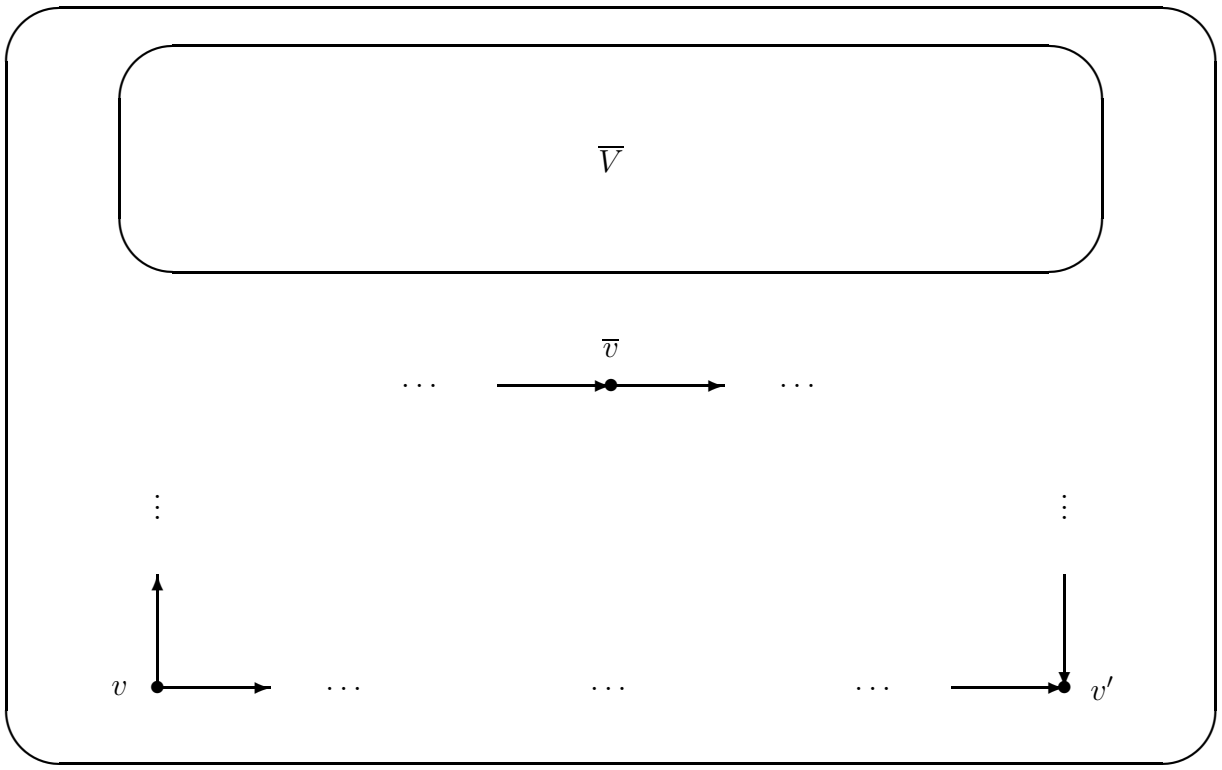
    Die erste Zeile ist klar, denn kürzeste Wege, die nichts vermeiden müssen, sind allgemeine kürzeste Wege. Die zweite Zeile ist nach dem vorigen Punkt korrekt. Der leere Weg ist immer ein kürzester Weg von v nach v, der zwischendurch keine Knoten besucht. Das erklärt die dritte Zeile. Die kürzesten Wege 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 $\overline{v}$ oder vermeidet diesen zusätzlich.



Kapitel 4

Minimale aufspannende Bäume

Das Problem der minimalen aufspannenden Bäume ist mit dem Kürzesten-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 Entfernungsfunktion. In den Anwendungen werden dabei die Knoten oft als lokale Sende- und Empfangsstationen oder Prozessoren gedeutet und die Kanten als direkte Nachrichtenkanäle. Die Entfernung mag dann konkret so etwas wie Mietkosten o. ä. repräsentieren. Die Aufgabe besteht dann darin, in einem solchen Kommunikationsnetz möglichst "kostengünstig" Nachrichten von einem Knoten zu allen anderen senden zu können. Minimiert werden soll dabei die Entfernungssumme 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 hingeschickt 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. Ein Graph $G = (V, E)$ ist *zusammenhängend*, wenn es je einen Weg von v nach v' für alle $v, v' \in V$ gibt.

2. 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.

3. 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.

4. 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.

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

   Dabei ist die *Entfernung* für $G' \subseteq G$ mit $G'=(V', E')$ gegeben durch $dist(G') = \sum_{e' \in E'} dist(e')$.

6. Der folgende Algorithmus konstruiert zu jedem zusammenhängenden Graphen und zu jeder Entfernungsfunktion einen minimalen aufspannenden Baum:

   min-spantree 1

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

7. Der Algorithmus **min-spantree 1** ist polynomiell.

Anhang zur Korrektheit und zum Aufwand

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. Dazu muss das Ergebnis des Verfahrens bezüglich der Entfernung mit einem beliebigen aufspannenden Baum B'' mit Kantenmenge E'' verglichen werden. Um das leichter tun zu können, werden die gelöschten Kanten der Reihenfolge des Löschens nach mit $e_1, \ldots, e_k$ benannt. Die in den k Schritten konstruierten Kantenmengen und Teilgraphen sind dann $E_1, \ldots, E_k$ und $G_1, \ldots, G_k$ mit $E_i = E_{i-1} - \{e_i\}$ und $G_i = G_{i-1} - e_i$ für $i = 1, \ldots, k$, wenn man $E_0 = E$ und $G_0 = G$ wählt. Nach Wahl der gelöschten Kanten gilt für $i = 1, \ldots, k$:

(a) $e_i \in E_{i-1} - BRIDGE(G_{i-1})$,
(b) $dist(e_i) \geq dist(e')$ für alle $e' \in E_{i-1} - BRIDGE(G_{i-1})$,
wobei (a) insbesondere bedeutet, dass es in $G_i = G_{i-1} - e_i$ einen Weg p_i gibt, der die beiden Knoten von e_i verbindet.
Die Situation lässt sich folgendermaßen skizzieren:

Außerdem ist G_k der vom Algorithmus konstruierte aufspannende Baum. Es ist also $dist(G_k) \leq dist(B'')$ zu zeigen.

Da B'' auch ein aufspannender Baum ist, können sich B'' und G_k nur in den Kantenmengen unterscheiden. Die von G_k ist bekannt: $E_k = E - \{e_1, \ldots, e_k\}$. Die von B'' ist unbekannt, aber es lassen sich zwei Fälle unterscheiden.

1. Fall: $E'' \subseteq E_k$. Angenommen, es gäbe ein $e_0 \in E_k - E''$. Da B'' zusammenhängend ist, existiert ein Weg p_0 in B'', der die Knoten von e_0 verbindet. Wegen $B'' \subseteq G_k$, liegt p_0 auch in G_k, so dass e_0 keine Brücke ist − im Widerspruch zur Haltebedingung des Algorithmus. Also ist $E'' = E_k$ und damit $B'' = G_k$ sowie insbesondere $dist(G_k) = dist(B'')$, wie gewünscht.

2. Fall: $E'' \not\subseteq E_k$. Dann existiert ein i_0 mit $e_{i_0} \in E''$, aber $e_i \notin E''$ für $i_1, \ldots, i_0 - 1$. Die Idee ist nun, e_{i_0} durch eine Kante aus E_k auszutauschen, so dass aus B'' wieder ein aufspannender Baum entsteht, ohne dass die Entfernung größer wird. Mit diesem neuen Baum wird wie mit B'' verfahren, bis alle falschen Kanten ausgetauscht sind. Es handelt sich tatsächlich um eine vollständige Induktion über die Zahl der Kanten aus $e_1, \ldots, e_k$, die in E'' liegen. Der 1. Fall ist der Induktionsanfang, der 2. Fall der Induktionsschluss, wobei der neue Baum gegenüber B'' gerade die Kante e_{i_0} nicht mehr enthält.

Bleibt der Austausch zu konstruieren. G_k ist zusammenhängend, so dass es einen Weg p_{i_0} in G_k gibt, der die Knoten von e_{i_0} verbindet. Da B'' ein Baum ist und somit e_{i_0} eine Brücke, kann p_{i_0} nicht ganz in B'' liegen. Es gibt also in p_{i_0} eine Kante $e_0 \in E_k - E''$. Da B'' aufspannend ist, gibt es in B'' einen einfachen Weg p_0, der die Knoten von e_0 verbindet. Liegt e_{i_0} nicht auf diesem Weg, kann man e_0 in p_{i_0} durch p_0 austauschen. Es entsteht ein Weg, der die Knoten von e_{i_0} verbindet, aber weniger Kanten aus $E_k - E''$ enthält als p_{i_0}. Nennt man den neuen Weg wieder p_{i_0} und wiederholt die obige Überlegung, trifft man irgendwann auf eine Kante $e_0 \in E_k - E''$ einen einfachen Weg p_0 in B'', der e_{i_0} enthält und die Knoten von e_0 verbindet. Denn sonst erhielte man doch einen Weg in B'', der die Knoten von e_{i_0} verbindet und selbst e_{i_0} nicht enthält, den es aber wegen der Brückeneigenschaft von e_{i_0} nicht geben kann. Somit gilt $p_0 = p' \bullet e_{i_0} \bullet p''$ für geeignete Teilwege p', p'', die e_{i_0} nicht enthalten. Eine Skizze zeigt die

erreichte Situation.

Nun kann e_{i_0} gegen e_0 ausgetauscht werden, was zu dem Teilgraphen $B' = (V, (E'' - \{e_{i_0}\}) \cup \{e_0\})$ führt. B' ist zusammenhängend. Denn für beliebige v, v' gibt es einen verbindenen Weg p in B'', der entweder auch in B' liegt oder die Kante e_{i_0} enthält. Die kann dann aber gegen den Weg $p' \bullet e_0 \bullet p''$, der in B' liegt, ausgetauscht werden. Ohne Beschränkung der Allgemeinheit kann man annehmen, dass alle Kanten von B' Brücken sind, denn Kanten, die keine Brücken sind, lassen sich nach und nach herausnehmen. Zusammenfassend ist B' also ein aufspannender Baum mit $dist(B') = dist(B'') - dist(e_{i_0}) + dist(e_0)$. Das impliziert $dist(B') \leq dist(B'')$, falls $dist(e_{i_0}) \geq dist(e_0)$. Das aber folgt aus der Eigenschaft (b) (siehe oben), falls $e_0 \in E_{i_0-1} - BRIDGE(G_{i_0-1})$. Das schließlich ergibt sich folgendermaßen: Nach Wahl von B'' gilt $e_i \notin E''$ für $i = 1, \ldots, i_0 - 1$ und somit $B'' \subseteq G_{i_0-1}$. Außerdem ist $e_0 \in E_k \subseteq E_{i_0-1}$, so dass der Weg $p' \bullet e_{i_0} \bullet p''$ die Knoten von e_0 in G_{i_0-1} verbindet, ohne e_0 zu enthalten. Also ist e_0 keine Brücke von G_{i_0-1}, was als letztes noch zu zeigen war.

Kapitel 5

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)* $cap : E \to \mathbb{N}$.

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

$$val(flow) = outflow(A) - inflow(A) = inflow(B) - outflow(B).$$

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:

$$val(flow) = outflow(A) = inflow(B).$$

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

5. Am Knoten A lässt sich ein Fluss $flow : E \to \mathbb{N}$ verbessern, wenn es gelingt, den Fluss einmündender Kanten zu verkleinern und den ausgehender zu vergrößern, und umgekehrt an 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:

   (i) $\xrightarrow{flow(e_1)+c} \bullet \xrightarrow{flow(e_2)+c}$
   (ii) $\xrightarrow{flow(e_1)+c} \bullet \xleftarrow{flow(e_2)-c}$
   (iii) $\xleftarrow{flow(e_1)-c} \bullet \xrightarrow{flow(e_2)+c}$
   (iv) $\xleftarrow{flow(e_1)-c} \bullet \xleftarrow{flow(e_2)-c}$

   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 $e \in E$ $\overline{e}$ eine sogenannte *Rückwärtskante*, sei $\overline{E} = \{\overline{e} \mid e \in E\}$, und bezeichne $\overline{G}$ den um die Rückwärtskanten erweiterten Graphen G, d.h. $\overline{G} = (V, E \cup \overline{E}, \overline{s}, \overline{t})$ mit $\overline{s}(e) = s(e)$ und $\overline{s}(\overline{e}) = t(e)$ sowie $\overline{t}(e) = t(e)$ und $\overline{t}(\overline{e}) = s(e)$ 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 $\overline{G}$ mit $cap(\hat{e}_i) - flow(\hat{e}_i) > 0$ für $\hat{e}_i \in E$ und $flow(\hat{e}_i) > 0$ für $\hat{e}_i \in \overline{E}$ $(i = 1, \ldots, n)$.

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

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

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* $zero : E \to \mathbb{N}$ mit $zero(e) = 0$ für alle $e \in E$ mit dem Durchsatz $val(zero) = 0$ verwendet werden kann, lässt 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. Jungnickel).

   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

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

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

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

   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:

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

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:

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

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:

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

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

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

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:

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

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:

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

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:

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

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 $s(e) \in S_0$ mit $t(e) \in T_0$. Dann gibt es einen aufsteigenden Pfad $\hat{e}_1 \cdots \hat{e}_n$ von A nach $s(e)$. Wäre $\mathit{flow}(e) < cap(e)$, 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 $\mathit{flow}(e) = cap(e)$. 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 $\hat{e}_1 \cdots \hat{e}_n \overline{e}$ 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:

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

Kapitel 6

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 $\sharp V - 1$ 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 $\sharp V - 1$ kann $\sharp V!$ 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.

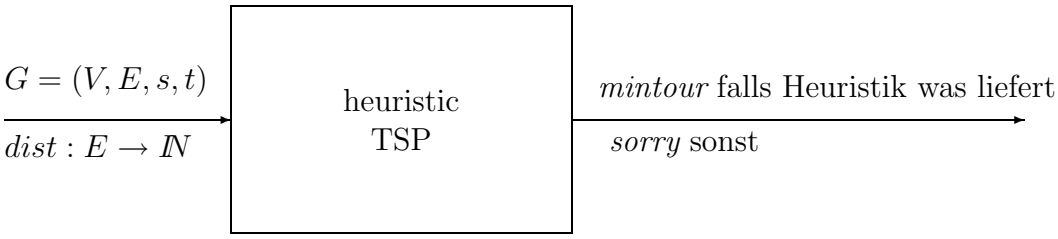
P: Scheint?

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 7

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 werde. Dafür einige Beispiele.

7.1 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 $dist : E \to \mathbb{N}$ 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 $i_1 \cdots i_n$ existiert mit $\{i_1, i_2\}, \ldots, \{i_{n-1}, i_n\}, \{i_n, i_1\} \in E$; *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 $color : V \to [k]$ 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 $color : V \to [k]$.

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

   nichtdeterministisches Verfahren: Rate für jeden Knoten die richtige Farbe.

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

   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 $K_m \subseteq G$, *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 $\left(\begin{array}{c} n \\ m \end{array} \right)$ (was für m in der Nähe von $\frac{n}{2}$ exponentiell ist).

   nichtdeterministisches Verfahren: Rate die Elemente von V'.

   Aufwand: höchstens n.

   Wie bekommt man die größte Clique oder zumindest ihre Knotenzahl?

7.2 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.

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

Es ist dagegen völlig unklar, ob $NP \subseteq P$ 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 $P = NP$ 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 $translate : IN \to IN'$ mit polynomiell beschränkter Berechnungslänge gibt, so dass gilt:

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

   Ein Entscheidungsproblem D lässt sich eineindeutig durch die Menge der positiv beantworteten Eingaben $L_D = \{x \in IN \mid D(x) = JA\}$ charakterisieren. Die Eigenschaft einer Reduktion lässt sich dann so ausdrücken:

   $$x \in L_D \quad \text{gdw.} \quad translate(x) \in L_{D'}.$$

   Statt L_D wird manchmal wieder D selbst geschrieben, so dass $x \in D$ und $D(x) = JA$ einerseits und $x \notin D$ und $D(x) = NEIN$ andererseits dasselbe bedeuten.

   Beispiele

HAM $\leq$ **TSP** und **TSP** $\leq$ **HAM**.

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**

$L_0 \in NP$ ist *NP-vollständig*, wenn $L \leq L_0$ für alle $L \in NP$.

Beobachtung

Sei L_0 *NP*-vollständig. Dann gilt:

(1) $L_0 \in P$ gdw. $P = NP$.

(2) $L_0 \leq L'$ und $L' \in NP$ impliziert L' *NP*-vollständig.

Beweis: (1) $P = NP$ impliziert $L_0 \in P$, denn nach Voraussetzung ist $L_0 \in NP$. Sei umgekehrt $L_0 \in P$. Dann ist $NP \subseteq P$ zu zeigen (da $P \subseteq NP$ klar ist). Sei dazu $L \in NP$. Dann gilt $L \leq L_0$, so dass L die sequentielle Komposition einer polynomiellen Übersetzung und des polynomiellen L_0 ist und somit selbst polynomiell.

(2) $L_0 \leq L'$ impliziert eine polynomielle Übersetzung $translate' : IN_0 \to IN'$, so dass gilt:

$$(*) \quad y \in L_0 \text{ gdw. } translate'(y) \in L'.$$

Sei nun $L \in NP$. Dann ist $L \leq L_0$, da L_0 NP-vollständig ist. Also existiert eine polynomielle Übersetzung $translate : IN \to IN_0$, so dass gilt:

$$(**) \quad x \in L \text{ gdw. } translate(x) \in L_0.$$

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

$$x \in L \text{ gdw. } translate(x) \in L_0 \text{ gdw. } translate'(translate(x)) \in L'.$$

Mit anderen Worten gilt: $L \leq L'$, und da L beliebig gewählt ist, erweist sich L' 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 $NP - P$ 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.

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 $\overline{x}$ für $x \in X$; $\overline{x}$ 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 $ass : X \to \{0, 1\}$, 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 $k = \sharp X$.
(2) k-mal Raten, d.h. **SAT** $\in$ *NP*.

Theorem: **SAT** ist *NP*-vollständig ist.

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

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

4. *NP*-**vollständige Graphenprobleme**

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

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

(2) **SAT3** $\leq$ **COLOR**, so dass auch **COLOR** *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 $f = c_\wedge \wedge \cdots \wedge c_n$ mit $c_i = l_{i1} \vee l_{i2} \vee l_{i3}$ in einen geeigneten Graph 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**.

Der Beweis für die zweite Reduktion ist ähnlich.

Kapitel 8

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.

8.1 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 wieweit 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.

8.2 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 werden, liefert sie wichtige Erkenntnisse. Das Optimum im erweiterten Lösungsraum ist nämlich eine untere Schranke für das eigentlich gesuchte Optimum. Man bekommt also eine Ahnung, wieweit 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 degenerierte – aufspannender Baum. Der Wert eines minimalen aufspannenden Baumes, der bekanntlich polynomiell bestimmt werden kann, bildet somit eine untere Schranke für die Entfernung von Rundreisen.

8.3 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 $dist : E \to \mathbb{N}$ 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:

$$bound(M, dist) = \sum_{v \in V} (\min_{e \in E}\{dist(e) | s(e) = v\} + \min_{e \in E}\{dist(e) | t(e) = v\})/2.$$

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.

8.4 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 Überpfüfung 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 kann. 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.

Kapitel 9

Reduktion von 3SAT auf CLIQUE

Das Erfüllbarkeitsproblem der Aussagenlogik besteht in der Frage, ob eine aussagenlogische Formel bei geeigneter Wahl der Wahrheitswerte für die Grundaussagen wahr werden kann. Mit Hilfe einer Wahrheitstafel lässt sich das überprüfen, indem alle Möglichkeiten durchprobiert werden. Das Erfüllbarkeitsproblem, das hier für Formeln in spezieller konjuktiver Normalform betrachtet wird, ist in der Informatik zu Berühmtheit gelangt, weil Cook 1971 zeigen konnte, dass es das schwierigste *NP*-Problem überhaupt ist. Dieses Ergebnis hat auch viele interessante Konsequenzen für Algorithmen auf Graphen. Hier wird vor allem eine Reduktion auf das Cliquenproblem konstruiert. Wozu das gut sein soll, wird hinten diskutiert.

1. Sei X eine Menge von *Grundaussagen*, die oft auch *Boolesche Variablen* genannt werden. Unter einem *Literal* versteht man eine Grundaussage und ihre Negation $\neg x$ für $x \in X$. Eine *Klausel* ist eine Disjunktion von Literalen (hier speziell immer drei). Eine *Formel* f ist eine Konjunktion von Klauseln, d.h.

   $f = c_1 \wedge \cdots \wedge c_m = (l_{11} \vee l_{12} \vee l_{13}) \wedge \cdots \wedge (l_{m1} \vee l_{m2} \vee l_{m3})$ für $m \in \mathbb{N}$.

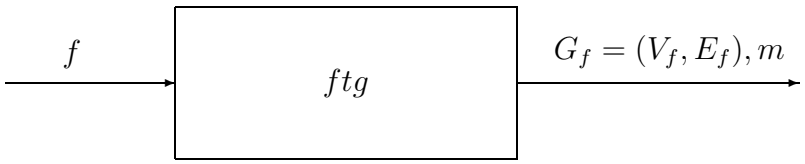
2. Eine Formel f lässt sich bezüglich einer *Belegung* der Grundaussagen $a : X \to BOOL$ folgendermaßen auswerten: $eval_a(f) = true$ gdw. $eval_a(c_i) = true$ für alle $i = 1, \ldots, m$ gdw. zu jedem $i = 1, \ldots, m$ ein $j(i)$ mit $eval_a(l_{ij(i)}) = true$ existiert, wobei für eine Negation $eval_a(\neg x) = \neg eval_a(x)$ und für eine Grundaussage $eval_a(x) = a(x)$ gilt.

3. Das *Erfüllbarkeitsproblem* (3SAT) ist nun für X und f als Eingabe die Frage, ob es eine Belegung $a : X \to BOOL$ mit $eval_a(f) = true$ gibt. Für die positive Antwort schreibt man kurz $(X, f) \in$ 3SAT bzw. $f \in$ 3SAT.

4. Offensichtlich gilt: **SAT3** $\in NP$, denn man kann einfach für jede Grundaussage die richtige Belegung (von zwei möglichen) raten.

5. Eine naheliegende deterministische Lösung besteht darin, alle Belegungen durchzuprobieren (Wahrheitstafel). Es gibt 2^n Belegungen, so dass dieser Algorithmus jedenfalls mindestens exponentiell ist. Allerdings ist das Überprüfen, ob eine Formel wahr wird bezüglich einer gegebenen Belegung a in linearer Zeit möglich bezogen auf die Zahl der Klauseln (oder Literale). Denn man muss die Klauseln einmal durchlaufen und unter den je drei Literalen nachsehen, ob ein $x \in X$ vorkommt mit $a(x) = true$ oder eine $\neg y$ für $y \in X$ mit $a(y) = false$.

6. $f \in$ **SAT3** bedeutet, dass man m Literale $l_{ij(i)}$ für $i = 1, \ldots, m$ finden kann (je ein Literal in jeder Klausel), so dass alle gleichzeitig wahr sein können. Betrachtet man die Literale als Knoten und zieht Kanten zwischen Literale, wenn sie gleichzeitig wahr sein können, bilden die m Literale oben eine Clique. Beachtet man dann noch, dass zwei Literale nur dann nicht gleichzeitig wahr sein können, wenn das eine x und das andere $\neg x$ ist, so erhält man folgende Konstruktion:



wobei $V_f = \{1, \ldots, m\} \times \{1, 2, 3\}$ und E_f eine Kante $\{(i, j), (\overline{i}, \overline{j})\}$ genau dann enthält, wenn gilt: $i \neq \overline{i}$ und $l_{ij} = x$ impliziert $l_{\overline{i}\,\overline{j}} \neq \neg x$.

7. Die Übersetzung ftg ist polynomiell, denn für je zwei Literale (insgesamt also nicht mehr als $(3m)^2$) muss überprüft werden, ob eins eine Grundaussage ist und das andere die Negation davon.

8. Die Übersetzung ist korrekt, d.h. es gilt: $f \in$ **SAT3** gdw. $(G_f, m) \in$ **CLIQUE**.

Das sieht man folgendermaßen ein: $f \in$ **SAT3** heißt, dass es zu jedem $i = 1, \ldots, m$ ein $j(i) \in \{1, 2, 3\}$ gibt mit $eval_a(l_{ij(i)}) = true$ für eine geeignete Belegung $a : X \to BOOL$. Dann bilden die Paare $(i, j(i))$ m Knoten. Betrachte zwei davon: $(i, j(i))$ und $(\overline{i}, j(\overline{i}))$ mit $i \neq \overline{i}$. Sei außerdem $l_{ij(i)} = x \in X$. Wäre $l_{\overline{i}j(\overline{i})} = \neg x$, dann ergäbe die Auswertung bzgl. a:

$$eval_a(l_{\overline{i}j(\overline{i})}) = eval_a(\neg x) = \neg eval_a(x) = \neg eval_a(l_{ij(i)}).$$

Das steht im Widerspruch dazu, dass beide Literale *true* als Auswertung haben. Also ist $l_{\overline{i}j(\overline{i})} \neq \neg x$, so dass $\{(i, j(i), (\overline{i}, j(\overline{i}))\} \in E_f$ und $U_a = \{(i, j(i)) | i = 1, \ldots, m\} \subseteq V_f$ eine m-elementige Clique in G_f bildet. Das bedeutet: $(G_f, m) \in$ **CLIQUE**.
Sei umgekehrt $(G_f, m) \in$ **CLIQUE**. Dann gibt es eine Teilmenge $U \subseteq V_f$ mit $\sharp U = m$ und $\binom{U}{2} \subseteq E_f$. Sei $U = \{v_1, \ldots, v_m\}$. Dann gibt es für jedes $k = 1, \ldots, m$ $i(k) \in \{1, \ldots, m\}$ und $j(k) \in \{1, 2, 3\}$ mit $v_k = (i(k), j(k))$. Für $k \neq \overline{k}$ ist $\{v_k, v_{\overline{k}}\} \in E_f$, so dass $i(k) \neq i(\overline{k})$ ist und $l_{i(\overline{k})j(\overline{k})} \neq \neg x$, falls $l_{i(k)j(k)} = x \in X$. Damit bilden insbesondere $i(1), \ldots, i(m)$ eine Permutation von $1, \ldots, m$, und von den Literalen $l_{i(1)j(1)}, \ldots, l_{i(m)j(m)}$ ist je eins aus jeder Klausel von f. Wenn es eine Belegung gibt, die diese m Literale gleichzeitig wahr macht, ist f erfüllt und damit, wie gewünscht, $f \in$ **SAT3**. Es bleibt, die Belegung zu konstruieren. Für $x \in X$ gibt es drei Fälle.
(1) Es existiert k mit $l_{i(k)j(k)} = x$. Dann wähle $a(x) = true$, und es gilt:

$$eval_a(l_{i(k)j(k)}) = eval_a(x) = a(x) = true.$$

(2) Es existiert $\overline{k}$ mit $l_{i(\overline{k})j(\overline{k})} = \neg x$. Dann wähle $a(x) = false$, und es gilt:

$$eval_a(l_{i(\overline{k})j(\overline{k})}) = eval_a(\neg x) = \neg eval_a(x) = \neg a(x) = \neg false = true.$$

(3) Ansonsten kommt x unter den speziellen Literalen nicht vor, so dass $a(x)$ beliebig gewählt werden kann.
Dieses a ist eine Abbildung von X nach $BOOL$, weil für kein x die beiden ersten Fälle gleichzeitig auftreten, wie oben gezeigt. Also ist a eine Belegung, und die Auswertungen in (1) und (2) sind wohldefiniert. Außerdem ist jedes der speziellen Literale entweder eine Grundaussage oder eine Negation davon, so dass jedes in (1) oder (2) vorkommt und durch a wahr wird.

Alles zusammen zeigt, dass die Übersetzung korrekt ist und somit wegen Punkt 7 eine Reduktion. D.h. **SAT3** $\leq$ **CLIQUE**. Mit **SAT3** erweist sich damit auch **CLIQUE** als NP-vollständig. Alle anderen *NP*-Probleme lassen sich also auf **CLIQUE** reduzieren.

9.1 Einige Ergebnisse über *NP*-Vollständigkeit

In der Klasse *NP* gibt es Probleme, die schwieriger sind als alle anderen.

$D \in NP$ ist NP-*vollständig*, falls $D' \leq D$ für alle $D' \in NP$. Die Klasse der *NP*-vollständigen Probleme wird mit NPV bezeichnet.

Schlüsselergebnis ist der Satz von Cook (1971), der zeigt, dass NPV nicht leer ist.

1. **SAT3** $\in NPV$.

Um $NP \subseteq P$ zu zeigen, reicht es, in P ein *NP*-vollständiges Problem zu finden.

2. $NPV \cap P \neq \emptyset$ impliziert $NP \subseteq P$.

Um ein *NP*-Problem als *NP*-vollständig zu erkennen, muss man ein anderes *NP*-vollständiges Problem darauf reduzieren.

3. $D \leq D'$ für $D \in NPV$ und $D' \in NP$ impliziert $D' \in NPV$.

Literaturverzeichnis

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

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

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

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

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

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

[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.