Graphentheorie für Studierende der Informatik

Christoph Klauck, Christoph Maas

6. Auflage 2015

Inhaltsverzeichnis

Kapitel 0

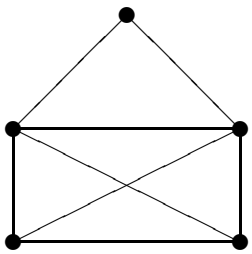
Einleitung

Graphentheorie

Die Graphentheorie ist ein Teilgebiet der Mathematik, dessen Anfänge bis ins 18. Jahrhundert zurückreichen (Leonhard Eulers „Königsberger Brückenproblem“ – s.u. Kapitel 5), das aber erst im 20. Jahrhundert größeres Interesse auf sich zu ziehen vermochte. Sie hat sich als nützliches Instrument zur Lösung verschiedenartigster Probleme erwiesen, etwa in den Wirtschaftswissenschaften, den Sozialwissenschaften oder der Informatik. Gerade für die Informatik ist die Graphentheorie ein wichtiges Gebiet, da dort Graphen einerseits zur rechnerinternen Repräsentation von Informationen und Daten häufig verwendet werden[1] und andererseits zur Visualisierung von bestimmten Sachverhalten dienen.

Aufgaben aus der Graphentheorie sind durch Denksportaufgaben weithin bekannt geworden, zum Beispiel:

1. Läßt sich das „Haus vom Nikolaus“ zeichnen, ohne einmal den Stift abzusetzen und ohne eine Linie doppelt zu zeichnen?



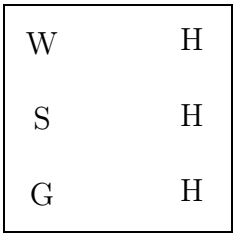
   Diese Aufgabe zeichnet sich dadurch aus, daß ein Problem in einem bestehenden Graphen gelöst wird. Etwas allgemeiner formuliert besteht das

[1] Stichwort: Abstrakter Datentyp, semantische Netze, Constraint-Netze etc.

Problem darin, alle Kanten eines gegebenen Graphen (hier das Haus des Nikolaus) in einer bestimmten Sequenz (in einem sogenannten Weg) genau einmal *abzulaufen*. Die visuelle Darstellung (Visualisierung) steht dabei im Hintergrund: Eine Lösung im Rechner könnte durchaus ohne Visualisierung, d.h. ohne Kenntnisse über Länge und Lage der Kanten bzw. Lage der Ecken, eine Lösung finden. Im Gegensatz dazu steht die visuelle Darstellung bei folgendem Problem im Vordergrund:

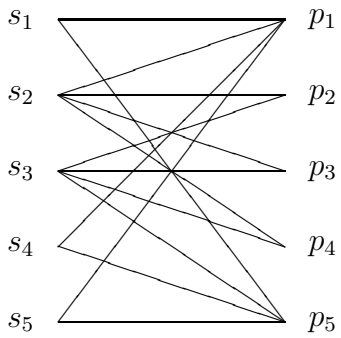
2. Auf einem Grundstück, auf dem drei Gebäude stehen, befindet sich je ein Anschluß für Wasser, Strom und Erdgas. Ist es möglich, jedes der Gebäude an jede der Versorgungseinrichtungen so anzuschließen, daß jede Anschlußleitung nur auf dem Grundstück verläuft und sich keine zwei Anschlußleitungen überschneiden?

   Hier ist eine Skizze – probieren Sie's aus!



   Hier ist zur Lösung des Problems die Kenntnis über die Lage der Ecken im Bild notwendig. Als Lösung wird ausschließlich der visuelle Verlauf der Kanten bestimmt; welche Ecken verbunden werden müssen, ist vorgegeben.

3. Von großer praktischer Bedeutung sind Methoden der Graphentheorie, wenn es darum geht, aus einer Menge diskreter (also wohlunterschiedener, nicht kontinuierlich ineinander übergehender) Handlungsmöglichkeiten eine optimale Handlung auszuwählen, wie etwa in dem folgenden Beispiel: Für die Semesterferien kann eine Firma für fünf verschiedene Projekte $p_1, \ldots, p_5$ je einen Informatikstudenten oder eine Informatikstudentin einstellen. Es bewerben sich insgesamt fünf Personen $s_1, \ldots, s_5$, die aber aufgrund ihrer jeweiligen Vorkenntnisse und Fähigkeiten immer nur für gewisse Projekte geeignet sind. Zur Entscheidungsfindung stellt der Abteilungsleiter die Liste der Personen und die Liste der Projekte einander gegenüber und verbindet eine Person und ein Projekt mit einer Linie, wenn die Person in diesem Projekt einsetzbar ist. Auf diese Weise erhält er folgendes Schema:



Er entscheidet sich daraufhin, drei Personen einzustellen, und zwar:

Person	für Projekt
s_1	p_5
s_2	p_1
s_3	p_3

Fallen Ihnen andere Lösungen ein, die mehr BewerberInnen berücksichtigen ?

Nachbemerkung

Der vorliegende Text ist als Begleitmaterial für Lehrveranstaltungen entstanden, die der Autor Christoph Maas für Studierende der Informatik an den Fachhochschulen Darmstadt und Hamburg gehalten hat. Deshalb orientierte sich der Stoffumfang an dem für eine Lehrveranstaltung verfügbaren Zeitrahmen. Durch die Übernahme der Veranstaltung im SS98 von dem Autor Christoph Klauck sind einige wenige Erweiterungen vorgenommen worden.

Zum weiteren Einlesen in diese Disziplinen, insbesondere auch in hier nicht angesprochene Teilgebiete können beispielsweise die folgenden Bücher dienen:

L. Collatz, W. Wetterling:	Optimierungsaufgaben, Berlin, Heidelberg, New York 1971
W. Domschke, A. Drexl:	Einführung in Operations Research, Berlin, Heidelberg New York 1991
M. R. Garey, D. S. Johnson:	Computers and Intractability, San Francisco 1979
R. Halin:	Graphentheorie, Darmstadt 1989
K. Neumann, M. Morlock:	Operations Research, München, Wien 1993
B. Owsnicki-Klewe:	Algorithmen und Datenstrukturen, Augsburg 1994
W. Reisig:	Petrinetze, Berlin, Heidelberg, New York 1986
W. Schneeweiß:	Grundbegriffe der Graphentheorie für praktische Anwendungen, Heidelberg 1985
E.L. Lawler, J.K. Lenstra, et al	The Traveling Salesman Problem, New Yorck, 1985
J. Clark, D.A. Holton:	Graphentheorie: Grundlagen und Anwendungen, 1Heidelberg, 1994
B. Baumgarten:	Petri-Netze, Heidelberg, 1996

Ein besonderer Dank gebührt an dieser Stelle unserem Kollegen Bernd Owsnicki-Klewe für die kritische Durchsicht des Manuskripts. Alle noch vorhandenen Fehler bleiben natürlich trotzdem das Verdienst der Autoren.

Hamburg, im Juni 1999 Christoph Klauck und Christoph Maas

Nachbemerkung Zwei

Innerhalb des Bologna-Prozesses wurde bei der Umstellung die Vorlesung „Graphentheorie“ zunächst aus dem Curiculum gestrichen (Jahr 2000). Die Erfahrungen zeigten jedoch, dass gerade für die angewandte Informatik die gesamte Theorie auf die diskrete Mathematik umgestellt werden sollte. So wurde die Graphentheorie in der Veranstaltung „Graphentheoretische Konzepte und Algorithmen“ im Jahre 2010 wieder in das Curiculum mit aufgenommen.

Dieses Skript erfährt daher nach einer ca. 10 jährigen Pause eine Aktualisierung und Anpassung an die geänderten Rahmenbedingungen. Auf Grund personeller Veränderungen werden diese Änderungen nur durch den Autor Christoph Klauck vorgenommen. Die Änderungen selbst sind aber oft durch

Anmerkungen der Studierenden motiviert, die bei der Durcharbeitung dieses Buches den Autor auf zahlreiche Stellen aufmerksam machten, die noch verbessert werden könnten. Ihnen gilt ein besonderer Dank!

Hamburg, im Juni 2011 Christoph Klauck

Kapitel 1

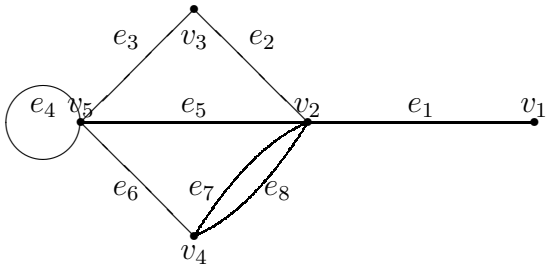
Grundbegriffe über Graphen

1.1 Gerichtete und ungerichtete Graphen

Definition 1.1 (Ungerichteter Graph) *Ein* Graph $G(V, E)$ *besteht aus einer nicht leeren Menge* V *von* Ecken *(englisch* vertices*) und einer Menge* E *von* Kanten *(englisch* edges*), die je zwei Ecken miteinander, oder eine Ecke mit sich selbst, verbinden. Die Ecken werden dabei z.B. mit* $v_1, v_2, \ldots, v_n$ *bezeichnet* $(n = |V|)$ *oder mit Buchstaben vom Ende des Alphabets, während die Kanten je nach Zweckmäßigkeit eigene Namen besitzen (z.B.* $e_1, e_2, \ldots, e_m$; $m = |E|$ *oder Buchstaben vom Anfang des Alphabets) oder durch die Ecken, die sie verbinden (die sogenannten* Endecken*), bezeichnet werden.*

Beim Zeichnen eines Graphen kommt es auf die geometrische Position der Ecken normalerweise nicht an. Man bemüht sich aber um eine möglichst übersichtliche Darstellung. Unter Umständen kann durch eine geschickte Drastellung die Charakteristik der durch den Graphen repräsentierten Beziehungen wesentlich verdeutlicht werden. Man denke z.B. hier an Programmablaufschemata, Prozeßdarstellungen im allgemeinen, ein S-/U-Bahn Netz etc.

Beispiel 1.1 *Dies ist ein Graph mit 5 Ecken und 8 Kanten*



Die Kante e_3 kann auch als v_3v_5 oder als v_5v_3 bezeichnet werden. Bei e_7 und e_8 kann diese Bezeichnung nicht verwendet werden, da sie in diesem Fall nicht eindeutig ist!

Man möge beachten, dass letztes Beispiel einen sogenannten zusammenhängenden Graphen zeigt; i.allg. kann ein Graph z.B. nur aus einer Menge von Ecken bestehen!

Definitionen können auch zur abkürzenden Schreibweise dienen, um in anderen Zusammenhängen sich umständliche Formulierungen zu ersparen. So auch folgende Definition, die eine Funktion einführt, die vorwiegend der abkürzenden Schreibweise dient.

Definition 1.2 *Sei $G = (V, E)$ ein ungerichteter Graph, dann bezeichne $s\_t : E \to 2^V$ die Menge*[1] *der durch eine Kante verbundenen Ecken.*

Beispiel 1.2 *Wir beziehen uns in diesem Beispiel auf den Graphen aus Beispiel 1.1. Es gilt u.a. $s\_t(e_1) = \{v_1, v_2\}$, $s\_t(e_4) = \{v_5\}$ und $s\_t(e_8) = \{v_2, v_4\}$.*

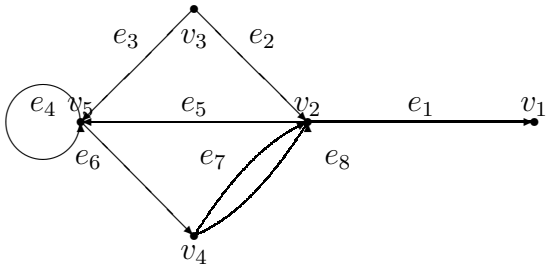
Nach Definition 1.1 sind beide Endecken einer Kante gleichberechtigt. Soll dies nicht der Fall sein, bezeichnet man eine der Ecken als *Anfangsecke* und die andere als *Endecke* der Kante und versieht bei einer Zeichnung das der Endecke zugewandte Ende der Kante mit einer Pfeilspitze. Neben dieser visuellen Änderung wird die Restriktion aktiviert, dass beim „Durchlaufen“ eines Graphen solche Kanten wie eine Einbahnstraße wirken: sie dürfen nur in einer Richtung, nämlich der Pfeilrichtung, durchlaufen werden.

Definition 1.3 (Gerichteter Graph) *Eine Kante, die genau eine Anfangsecke und eine Endecke besitzt, heißt eine* gerichtete Kante *(oder ein Pfeil, englisch* arc*). Ein Graph, dessen Kanten sämtlich gerichtet sind, heißt ein* gerichteter Graph *(oder ein Digraph*[2]*). Wenn gerichtete Kanten durch Eckenpaare bezeichnet werden, wird die Anfangsecke stets zuerst genannt.*

Beispiel 1.3 *Durch Richten aller Kanten des obigen Graphen entsteht beispielsweise der folgende Digraph:*

[1] 2^V bezeichnet die Potenzmenge über V; benötigt werden aber nur zwei- und einelementige Mengen.

[2] Aus dem Englischen übernommene Kurzform von „directed graph“.



Auch für gerichtete Graphen lassen sich analog zu Definition 1.2 nachfolgende Bezeichnungen festlegen.

Definition 1.4 *Sei $G = (V, E)$ ein gerichteter Graph, dann bezeichne*[3] *$s, t : E \to V$ die Ecke, die für eine Kante Anfangsecke (s) bzw. Endecke (t) ist.*

Beispiel 1.4 *Wir beziehen uns diesmal auf den Graphen aus Beispiel 1.3. Es gilt u.a. $s(e_1) = v_2$, $t(e_1) = v_1$, $s(e_4) = v_5$, $t(e_4) = v_5$ und $s(e_8) = v_4$, $t(e_8) = v_2$.*

Definition 1.5 *Sei G ein gerichteter Graph. Der Graph H, der entsteht, wenn alle gerichteten Kanten von G durch ungerichtete Kanten ersetzt werden, heißt* der G zugrundeliegende ungerichtete Graph.

Ein Graph, der sowohl gerichtete als auch nicht gerichtete Kanten besitzt, wird als gemischter Graph *bezeichnet.*

Definition 1.6 (Adjazenz und Inzidenz) *Zwei Ecken, die durch eine Kante verbunden sind, heißen* adjazent *(oder benachbart). Wenn v eine (Anfangs- oder) Endecke der Kante e ist, heißen v und e* inzident.

Definition 1.7 (Multigraph, schlichter/einfacher Graph)

1. *Eine Menge $\mathcal{E} \subseteq E$ von Kanten, deren Anfangs- und Endecken übereinstimmen, d.h. $\forall e_1, e_2 \in \mathcal{E} : s\_t(e_1) = s\_t(e_2)$, bzw. im Falle gerichteter Kanten $\forall e_1, e_2 \in \mathcal{E} : s(e_1) = s(e_2) \wedge t(e_1) = t(e_2)$, werden als* Mehrfachkanten *oder auch* parallele Kanten *bezeichnet. Gilt im Falle von gerichteten Kanten $\forall e_1, e_2 \in \mathcal{E} : s(e_1) = t(e_2) \wedge t(e_1) = s(e_2)$, so werden diese als* antiparallele Kanten *oder als* inverse Kanten *bezeichnet, jedoch nicht als Mehrfachkante.*

2. *Eine Kante $e \in E : s(e) = t(e)$ bzw. $|s\_t(e)| = 1$ heißt eine* Schlinge.

3. *Ein Graph mit Mehrfachkanten heißt ein* Multigraph.

[3] s bzw. t stehen für die englischen Begriffe *source* und *target*.

4. *Ein Graph ohne Schlingen und Mehrfachkanten heißt ein* schlichter Graph *oder auch* einfacher Graph.

Im Beispiel 1.1 sind die Ecken v_2 und v_4 durch eine Mehrfachkante miteinander verbunden und v_5 ist Endecke einer Schlinge.

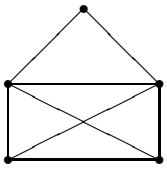
In vielen durch Graphen beschreibbaren Situationen können wir voraussetzen, dass wir es mit einfachen Graphen zu tun haben.

Definition 1.8 (Eckengrad) *Sei $v \in V$ eine Ecke eines Graphen $G = (V, E)$.*

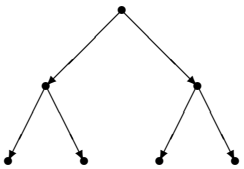
1. *Falls G ungerichtet ist, ist die Zahl $d(v)$ definiert als $d(v) = |\{e \in E \| v \in s\_t(e)\}| + |\{e \in E \| v \in s\_t(e) \wedge |s\_t(e)| = 1\}|$, d.h. die Anzahl der Kanten, deren Endecke v ist (dabei werden Schlingen doppelt gezählt, was durch den zweiten Summanden zum Ausdruck kommt!). $d(v)$ heißt* Grad der Ecke v.

2. *Falls G gerichtet ist, ist die Zahl $d_-(v)$ [bzw. $d_+(v)$] definiert als $d_+(v) = |\{e \in E \| s(e) = v\}|$ bzw. $d_-(v) = |\{e \in E \| t(e) = v\}|$, d.h. die Anzahl der Kanten, deren Ausgangsecke [bzw. Endecke] v ist. $d_+(v)$ [bzw. $d_-(v)$] heißt* Ausgangsgrad *[bzw.* Eingangsgrad*] der Ecke v.*

Beispiel 1.5 *1. Dieser Graph hat Ecken mit Graden 2,3 und 4:*



2. In diesem Digraphen haben alle Ecken den Ausgangsgrad 2 oder 0 und den Eingangsgrad 1 oder 0:



Satz 1.1 *Für die Eckengrade eines ungerichteten Graphen $G(V, E)$ gilt:*

$$\sum_{v \in V} d(v) = 2 \cdot |E|$$

Beweis: Beweis durch vollständige Induktion über $|E|$: Im Fall $|E| = 0$ (d.h. der Graph besitzt keine Kanten) ist die Behauptung richtig. Jedes Hinzufügen einer Kante in einen (ungerichteten) Graphen erhöht die Summe der Eckengrade um 2, die Anzahl der Kanten aber um 1. q.e.d.

1.2 Teilgraphen und Untergraphen

Folgende Begriffe werden eingeführt, um bei gewissen Fragestellungen auch nur Teile von einem gegebenem Graphen betrachten zu können.

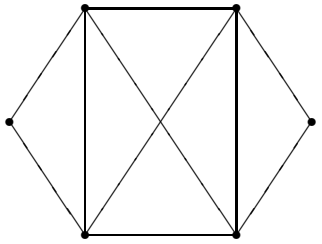
Definition 1.9 (Teilgraph) *Sei $G(V, E)$ ein Graph. Jeder Graph $H(W, F)$ mit $W \subseteq V$ und $F \subseteq E$ heißt ein* Teilgraph *von G.*

Ein Teilgraph entsteht also aus G durch Entfernen einiger Ecken (einschließlich der mit ihnen inzidenten Kanten) und Kanten[4].

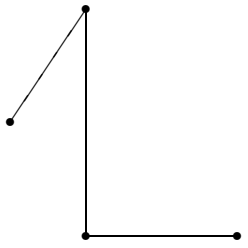
Definition 1.10 (Untergraph) *Ein Graph $H(W, F)$ mit $W \subseteq V$ heißt ein* Untergraph *von $G(V, E)$, wenn seine Kantenmenge F genau diejenigen Kanten aus E enthält, die zwei Ecken aus W verbinden.*

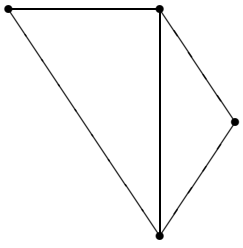
Ein Untergraph entsteht also aus G durch Entfernen einiger Ecken (einschließlich der mit ihnen inzidenten Kanten). Untergraphen sind somit spezielle Teilgraphen, da bei Teilgraphen die gleichen Manipulationen vorgenommen werden können. Teilgraphen sind aber i.allg. keine Untergraphen, da ein Teilgraph z.B. nur durch Entfernung von Kanten entstehen kann.

Beispiel 1.6 *1. Für den Graphen*

$G =$ 

sind beide folgenden Graphen H_1, H_2 Teilgraphen, aber nur H_2 ist auch ein Untergraph von G:

$H_1 =$ 

$H_2 =$ 

[4]Das Entfernen von Ecken impliziert das Entfernen der damit inzidenten Kanten, da sonst das Resultat kein Graph wäre; in einem Graphen verbindet eine Kante immer zwei Ecken bzw. eine Ecke mit sich selbst!

2. *Wenn das Straßennetz einer Stadt so als Graph aufgefaßt wird, dass jede Kreuzung oder Einmündung durch eine Ecke und jedes Straßenstück zwischen zwei (im Verlauf dieser Straße unmittelbar aufeinanderfolgenden) Ecken durch eine Kante dargestellt wird, dann ist das Netz aller vierspurig ausgebauten Straßen ein Teilgraph hiervon, während das Straßennetz eines Stadtteils darüber hinaus auch ein Untergraph ist.*

Aufgaben

1. Das europäische Luftverkehrsnetz werde durch einen Graphen dargestellt, dessen Ecken die Flughäfen repräsentieren und in dem Eckenpaare durch Mehrfachkanten verbunden werden (eine Kante für jede Fluglinie, die diese beiden Städte durch Nonstop-Flüge verbindet). Sind die folgenden Teilgraphen auch Untergraphen?

   (a) Die deutschen Flughäfen und die deutschen Inlandsflugverbindungen;

   (b) das Streckennetz einer bestimmten Fluglinie.

2. Zeichnen Sie einen Graphen mit vier Ecken, die die Grade 1, 2, 3 und 4 haben.

   Warum gibt es keinen einfachen Graphen mit dieser Eigenschaft ?

3. **Durchmesser** (? Punkte)
   Wir betrachten die Graphen G_n und H_n, deren Eckenmenge V alle n-stelligen Dezimalzahlen (erste Stelle ungleich Null) sind.

   - Zwei Zahlen sind adjazent in G_n, wenn sie sich an genau einer Stelle unterscheiden.
   - Zwei Zahlen sind adjazent in H_n, wenn sie sich an genau einer Stelle unterscheiden und der Unterschied zwischen den beiden Ziffern an dieser Stelle genau Eins ist.

   (a) Wie viele Ecken und wie viele Kanten haben die Graphen G_1 und G_2. Wie viele Kanten hat der Graph G_n allgemein (kurze Begründung) ?

   (b) Wie viele Ecken und wie viele Kanten haben die Graphen H_1 und H_2. Wie viele Kanten hat der Graph H_n allgemein (kurze Begründung) ?

(c) Den größten Abstand zwischen zwei Ecken in einem Graphen G bzgl. des kürzesten Wegs zwischen den beiden Ecken nennt man den **Durchmesser** $D(G)$ des Graphen. Welche Durchmesser haben die Grahen G_n und H_n ?

4. **Eckengrad** (? Punkte) Wir betrachten ungerichtete zusammenhängende schlichte Graphen:

   (a) Gibt es einen Graphen mit den Eckengraden $3; 3; 3; 3; 5; 6; 6; 6; 6; 6; 6$?

   (b) Gibt es einen Graphen mit den Eckengraden $1; 1; 3; 3; 3; 3; 5; 6; 8; 9$?

   (c) Gibt es einen bipartiten Graphen mit den Eckengraden $3; 3; 3; 3; 3; 5; 6; 6; 6; 6; 6; 6; 6; 6$?

1.3 Isomorphie von Graphen

In bestimmten Anwendungen der Informatik besteht die Aufgabe darin, einen vorgegebenen Graphen (im Sinne eines vorgegebenen Musters) in einem anderen Graphen zu finden bzw. einen ähnlichen Graphen zu finden. Basis dafür ist zunächst folgende Definition.

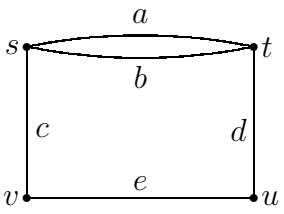
Definition 1.11 *Zwei Graphen $G(V, E)$ und $H(W, F)$ heißen* isomorph *(geschrieben: $G \cong H$), wenn es zwei bijektive Abbildungen*

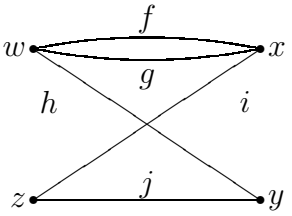
$$\varphi : V \to W, \quad \psi : E \to F$$

gibt, so dass für alle $u, v \in V$ und $e \in E$ gilt:

$$e = uv \Longleftrightarrow \psi(e) = \varphi(u)\varphi(v)$$

Beispiel 1.7 *1. Die folgenden beiden Graphen sind isomorph:*

$G =$ 

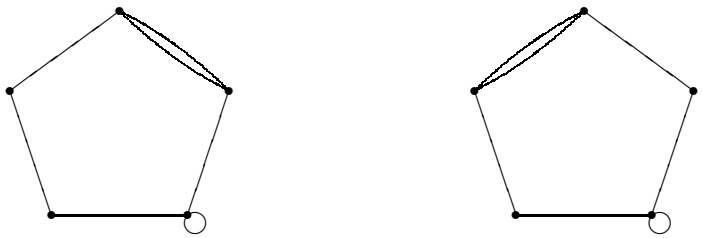
$H =$ 

Die geforderten bijektiven Abbildungen können beispielsweise sein:

$$\begin{array}{lcl} \varphi(s) & := & w \\ \varphi(t) & := & x \\ \varphi(u) & := & z \\ \varphi(v) & := & y \end{array} \quad , \quad \begin{array}{lcl} \psi(a) & := & g \\ \psi(b) & := & f \\ \psi(c) & := & h \\ \psi(d) & := & i \\ \psi(e) & := & j \end{array}$$

Interessierte LeserInnen mögen zur Übung des Nachweises der Isomorphie weitere geeignete Abbildungen angeben.

2. *Diese beiden Graphen sind nicht isomorph (Begründung?):*



Anschaulich kann die Isomorphie zweier Graphen auch so definiert werden: Zwei Graphen heißen isomorph, wenn sie sich nur in der Benennung ihrer Ecken und Kanten unterscheiden. Die unterschiedlichen visuellen Darstellungen eines Graphen sind natürlich auch paarweise isomorph.

Bei einigen Problemstellungen ist es teilweise sinnvoll, eine weniger strenge Assoziation als die Isomorphie zwischen Graphen zuzulassen. In diesem Fall können z.B. Ecken bzw. Kanten miteinander identifiziert werden. Beides läßt man normalerweise nicht zu, da dann z.B. ein ziemlich großer Graph mit einem Graphen assoziiert werden kann, der aus einer Ecke und einer Schleife besteht. Meist trifft man die Identifizierung von Ecken an. Man spricht in diesem Fall nur von homomorphen Graphen, da die zugrunde liegenden Abbildungen nur noch Homomorphismen sind, also keine Isomorphismen[5]

Aufgaben

1. Geben Sie 10 nichtisomorphe ungerichtete Graphen mit 4 Ecken und 4 Kanten an.

[5]Zur Erinnerung: Ein Homomorphismus F (lineare Abbildung, d.h. z.B. $F \circ s = s \circ F$) $F : A \to B$ ist: Monomorphismus, falls F injektiv, also $F(a) = F(a') \to a = a'$; Epimorphismus, falls F surjektiv, also $F(A) = B$; Isomorphismus, falls F bijektiv; Endomorphismus, falls $A = B$; Automorphismus, falls $A = B$ und F injektiv.

2. Warum kann es keinen ungerichteten Graphen geben, bei dem genau eine Ecke v einen ungeraden Grad $d(v)$ besitzt?

3. Sind diese beiden Graphen isomorph?

4. **Isomorphie** (? Punkte)
Welche der in Abbildung 1.1 (Seite 21) dargestellten Paare von gerichteten Graphen sind isomorph ? Für die Paare, die isomorph sind, schreiben Sie bitte einen Isomorphismus auf. Für die Paare, die nicht isomorph sind, ist zu erklären/begründen, warum sie es nicht sind.

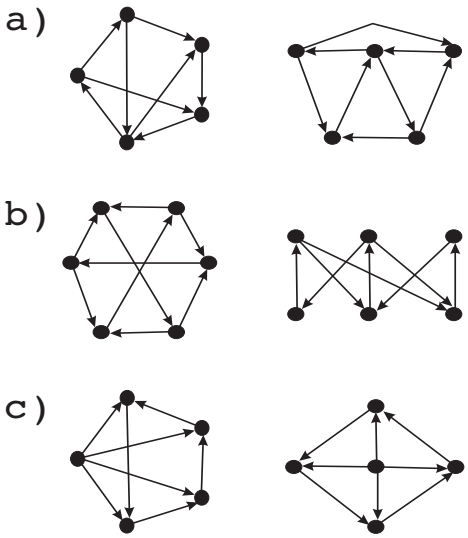


Abbildung 1.1: Welche von diesen Paaren sind isomorph ?

5. **Eckengrad** (? Punkte)
Zeichnen Sie einen zusammenhängenden Digraphen mit vier Ecken, in dem alle Ecken den Eingangsgrad zwei und den Ausgangsgrad zwei haben.

6. **Eckengrad** (? Punkte)
   Beweisen Sie mittels vollständiger Induktion das sogenannte „handshaking lemma“:

   Lemma 1.2 *Für die Eckengrade eines ungerichteten Graphen $G(V, E)$ gilt:*

   $$\sum_{v \in V} d(v) = 2 \cdot |E|$$

   Ermitteln Sie dann mittels dieses Lemmas die minimale Anzahl an Ecken eines vollständigen Graphen K_n, der mindestens 15 Kanten hat.

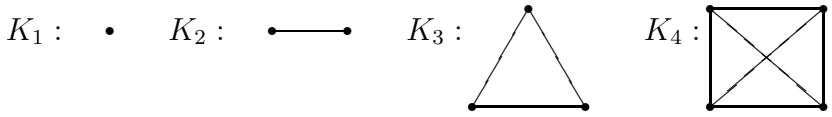
7. **Eckengrad** (? Punkte)
   Sieben Städte sollen durch Fluglinien miteinander verbunden werden und von jeder Stadt aus sollen genau drei andere Städte im (ungerichteten) Direktflug erreichbar sein. Geben Sie einen Plan an oder erklären Sie ggf. warum er nicht machbar ist.

1.4 Vollständige, leere und bipartite Graphen

Für bestimmte Mengen von Graphen haben sich eigene Bezeichnungen eingebürgert. Hier seien im nachfolgenden einige der verbreitesten aufgeführt.

Definition 1.12 *Ein einfacher Graph mit n Ecken heißt* vollständig*, wenn je zwei Ecken durch eine Kante verbunden sind. Er wird dann mit dem Symbol K_n bezeichnet.*

Beispiel 1.8 *Die ersten vier vollständigen Graphen sind:*



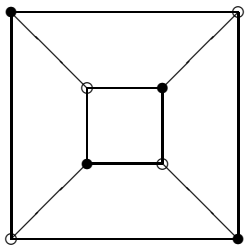
Der K_n mit n Ecken hat $\frac{n \cdot (n-1)}{2}$ Kanten.

Definition 1.13 *Ein Graph $G(V, E)$ mit $E = \emptyset$ heißt* leerer Graph. *$G(V, E)$ mit $V = E = \emptyset$ heißt* Nullgraph[6].

[6] Achtung: Nach unserer Definition 1.1 auf Seite 13 ist der Nullgraph kein Graph!

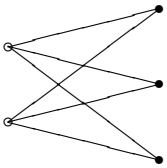
Definition 1.14 *Ein ungerichteter Graph $G(V, E)$ heißt* bipartit*, wenn sich seine Eckenmenge V derart in zwei disjunkte Teilmengen X, Y zerlegen läßt ($V = X \cup Y$; $X \cap Y = \emptyset$), dass jede Kante von G genau eine Endecke in X und eine Endecke in Y besitzt.*

Beispiel 1.9 *In dem folgenden bipartiten Graphen sind zur Verdeutlichung die Elemente der beiden Teilmengen von Ecken unterschiedlich dargestellt:*



Definition 1.15 *Ein schlichter bipartiter Graph mit $|X| = m$, $|Y| = n$, bei dem jede Ecke aus X mit jeder Ecke aus Y durch eine Kante verbunden ist, heißt ein* vollständiger bipartiter Graph *und wird mit dem Symbol $K_{m,n}$ bezeichnet.*

Beispiel 1.10 *Der vollständige bipartite Graph $K_{2,3}$ hat (abgesehen von isomorphen Darstellungen) folgende Gestalt:*



Aufgaben

1. Der Graph K_2 ist *Teilgraph* jedes Graphen $G = (V, E) : |E| > 1$. Geben Sie eine Klasse von Graphen an, für die der K_2 ein *Untergraph* ist.

2. **Vollständige Graphen** (? Punkte)
   Wir betrachten im Folgenden die vollständigen Graphen K_n.

   (a) Zeigen Sie für $2 \leq n \in \mathbb{N}$: Löscht man beim vollständigen Graphen K_n eine beliebige Ecke, so entsteht der vollständige Graph K_{n-1}.

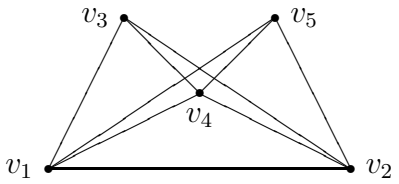
   (b) Zeigen Sie mittels vollständiger Induktion: $|E_{K_n}| = 0,5 * n * (n-1)$

1.5 Planare Graphen

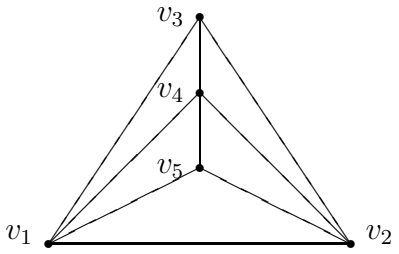
In diesem Abschnitt werden Begriffe eingeführt, die im Umfeld der visuellen 2D-Darstellung von Graphen verwendet werden.

Definition 1.16 *Ein Graph $G(V, E)$ heißt* planar*, wenn er in der Ebene so gezeichnet werden kann, dass jeder Punkt, den zwei Kanten gemeinsam haben, eine Ecke ist.*

Beispiel 1.11 *Bei dem folgenden Graphen ist zunächst nicht klar, ob er planar ist. Klar ist, dass diese visuelle Darstellung keine planare Darstellung ist: die Kante, die die beiden Ecken v_3 und v_4 verbindet, kreuzt sich mit zwei weiteren Kanten. Die Kreuzungspunkte sind jedoch keine Ecken!*

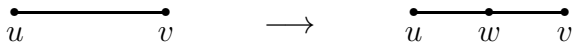


Eine andere Darstellung zeigt aber, dass es in der Tat möglich ist, ihn so zu zeichnen, dass sich keine Kanten außerhalb von Ecken berühren oder überschneiden:



Die planaren Graphen lassen sich auf eine einfache Weise charakterisieren (auch wenn der Beweis dieser Charakterisierung nicht ganz einfach ist und deshalb hier weggelassen wird):

- Es ist klar, dass ein Graph nicht planar sein kann, wenn er einen nichtplanaren Teilgraphen enthält.
- Weiterhin hat es keinen Einfluß auf die Planarität eines Graphen, wenn eine Kante eines Graphen durch Einfügen einer neuen Ecke mit Grad 2 in zwei Kanten zerlegt wird (der so entstehende Graph heißt eine Unterteilung des ursprünglich vorgelegten Graphen):

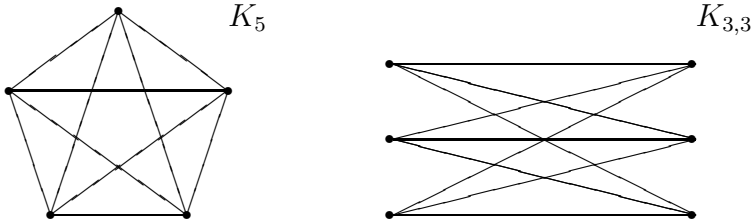


Eine Unterteilung des Graphen K_2

Im Lichte dieser Überlegungen stellt der folgende Satz fest, dass es „im wesentlichen“ nur zwei nichtplanare Graphen gibt. Dies hat ziemliche praktische Relevanz: der/die LeserIn möge sich vorstellen, die Planarität eines Graphen mit 10^{42} Kanten zu prüfen. Nachfolgender Satz gibt quasi einen Algorithmus vor: Prüfe, ob einer der beiden Graphen als Teilgraph enthalten ist. Sicherlich, dieses Problem ist immer noch schwierig, genau genommen NP-vollständig (vgl. Seite 47).

Satz 1.3 (Satz von Kuratowski) *Ein Graph $G(V, E)$ ist genau dann nichtplanar, wenn der zugrundeliegende ungerichtete Graph einen Teilgraphen besitzt, der isomorph ist zu*

1. *dem Graphen K_5 oder*
2. *dem Graphen $K_{3,3}$ oder*
3. *einer Unterteilung[7] des $K_{3,3}$ oder des K_5.*



Das Vierfarbenproblem

Die wohl bekannteste Aufgabenstellung im Zusammenhang mit planaren Graphen ist das Vierfarbenproblem:

Es ist eine Erfahrungstatsache, dass auf jeder Landkarte, die nur zusammenhängende Gebiete enthält (bei der also keine Exklaven zu berücksichtigen sind), die Gebiete so mit vier Farben gefärbt werden können, dass niemals zwei Gebiete, die eine gemeinsame Grenze (die nicht nur aus einem Punkt besteht) besitzen, dieselbe Farbe tragen. Stellt man jedes Gebiet durch eine Ecke dar

[7] Unterteilung heißt Unterteilung von Kanten durch beliebig viele Ecken, so dass die unterteilten Kanten sich nicht schneiden, also keine gemeinsame Ecke haben bzw. ein sogenannter Minor gebildet werden kann.

und verbindet man Gebiete mit gemeinsamer Grenze durch eine Kante, so entsteht ein planarer Graph. Läßt sich nun die genannte Erfahrungstatsache auch allgemein für planare Graphen beweisen oder beruht sie nur darauf, dass beim Färben bisheriger Landkarten glückliche Umstände mit im Spiel waren?

Satz 1.4 (Vierfarbensatz) *In jedem planaren Graphen $G(V, E)$ lassen sich die Ecken durch je eine von vier Farben so färben, dass keine zwei gleichfarbigen Ecken durch eine Kante miteinander verbunden sind.*

Es kann (vergleichsweise) leicht bewiesen werden, dass man zu einer solchen Färbung höchstens fünf Farben benötigt. Zum Nachweis, dass man tatsächlich bereits mit vier Farben zum Ziel kommt, veröffentlichten *Kenneth Appel* und *Wolfgang Haken* im Jahre 1976 eine Argumentation, von der aber lange Zeit noch nicht klar war, ob sie als Beweis akzeptiert werden kann, da sie auf einer sehr großen Zahl von mit dem Computer durchgeführten Fallunterscheidungen basiert und es zunächst nicht möglich war, alle diese Fälle „per Hand“ nachzuvollziehen[8].

Aufgaben

1. Machen Sie sich klar, dass die beiden im Satz von Kuratowski genannten Graphen tatsächlich nichtplanar sind und dass jeder ihrer Teilgraphen planar ist.

2. Warum ist es nicht möglich, die Staaten Europas mit je einer von drei Farben so zu färben, dass keine zwei benachbarten Länder dieselbe Farbe besitzen?

1.6 UND/ODER-Graphen

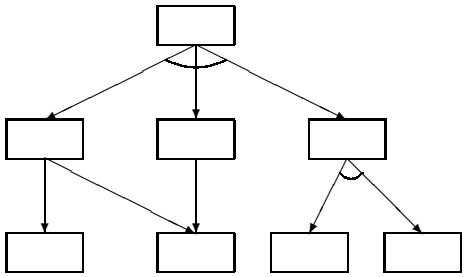
Es ist üblich, die Verwendbarkeit von Graphen für die Modellierung unterschiedlichster Sachverhalte dadurch zu erweitern, dass den Ecken oder Kanten bestimmte Werte bzw. Markierungen zugewiesen werden dürfen. So werden wir etwa in den folgenden Kapiteln Situationen begegnen, in denen Kanten eines Graphen eine „Länge“ zugeordnet wird. Als Beispiel dafür, wie sich durch diese Erweiterung des Graphenkonzepts neue Ausdrucksmöglichkeiten ergeben,

[8] vgl. *Appel,K. and W. Haken:* The Four Color Proof Suffices, Math. Intell. 8/1(1986), 10-20, *Sachs,H:* Einige Gedanken zur Geschichte und zur Entwicklung der Graphentheorie, Mitt. Math. Ges. Hamburg, XI/6(1989), 623-641 und *Horgan,J:* Der Tod des Beweises, Spektrum der Wissenschaft, 12/1993, 88-101

soll hier eine einfache Form von UND/ODER-Graphen vorgestellt werden, die im Bereich der KI z.B. zur Darstellung logischer Abhängigkeiten Verwendung finden.

Definition 1.17 *Sei $A := N \cup T$ ein Alphabet aus Nichtterminalzeichen und Terminalzeichen. Ein gerichteter Graph $G(V, E)$ heißt ein* **UND/ODER-Graph***, wenn jeder Ecke $v \in V$ eine Zeichenkette aus A^* sowie eines der Symbole UND, bzw. ODER zugewiesen worden ist.*

In der zeichnerischen Darstellung des Graphen werden UND-Ecken dadurch gekennzeichnet, dass alle ausgehenden Kanten durch einen Bogen miteinander verbunden werden. Die übrigen Ecken sind dann ODER-Ecken; also beispielsweise:



Ein Beispiel für eine Aufgabenstellung, die mit UND/ODER-Graphen behandelt werden kann, ist die Frage nach der Ableitbarkeit von bestimmten Zeichenketten in kontextfreien Sprachen.

Beispiel 1.12 *Sei $N := \{A, B, C, D, E, F\}$ und $T := \{\alpha\}$ mit den Ableitungsregeln*

$$\begin{aligned} A &\longrightarrow DE \\ A &\longrightarrow B\alpha \\ B &\longrightarrow \alpha\alpha \\ C &\longrightarrow BB\alpha \\ C &\longrightarrow F\alpha \\ D &\longrightarrow \varepsilon \\ E &\longrightarrow D\alpha \end{aligned}$$

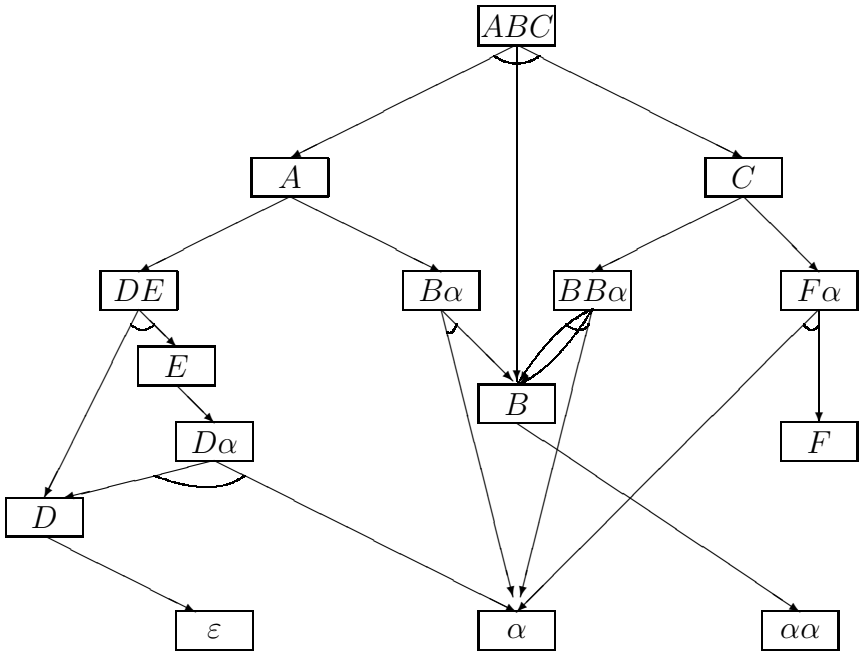
Kann die Zeichenkette ABC in eine Zeichenkette aus α^* überführt werden? Ein UND/ODER-Graph bietet eine übersichtliche Darstellung aller Ableitungsmöglichkeiten aus der Zeichenkette ABC und kann daher zur Lösung herangezogen

werden. Dabei stellen die gerichteten Kanten Manipulationen von Zeichenketten dar.

Wenn eine Zeichenkette durch Anwendung von einer der Regeln umgeformt wird, wird sie durch eine ODER-Ecke dargestellt. Wenn die Umformung in einer Zerlegung in Teilketten besteht, wird sie durch eine UND-Ecke dargestellt. Zeichenketten aus α^* werden nicht mehr weiter umgeformt.

Eine UND-Ecke kann in eine Zeichenkette aus α^* überführt werden, wenn alle *Endecken der von ihr ausgehenden Kanten in eine solche Zeichenkette überführt werden können. Eine ODER-Ecke kann in eine Zeichenkette aus α^* überführt werden, wenn sie selbst bereits eine solche Zeichenkette darstellt oder wenn (mindestens)* eine *Endecke einer von ihr ausgehenden Kante in eine solche Zeichenkette überführt werden kann.*

Ausgehend von den Zeichenketten ε, α und $\alpha\alpha$ läßt sich durch Rückwärtsverfolgen der gerichteten Kanten nachweisen, dass alle vorkommenden Zeichenketten bis auf $F\alpha$ und F in Zeichenketten aus α^* überführt werden können.



Weitere Anwendungsbeispiele sind etwa das Auffinden von Synthesemethoden für bestimmte chemische Substanzen (die Ecken stellen chemische Verbindungen dar; terminal sind dabei genau die verfügbaren Ausgangsprodukte; die Kanten sind von einer Verbindung zu ihren Vorstufen gerichtet; eine Ecke ist eine UND-Ecke, wenn sie aus *allen* mit ihr durch von ihr ausgehende

Kanten verbundenen Ecken gemeinsam synthetisiert werden kann; sie ist eine ODER-Ecke, wenn sie aus *jeder* mit ihr durch eine von ihr ausgehende Kante verbundenen Ecke allein synthetisiert werden kann) oder das Umformen eines unbestimmten Integrals in eine in einer Integraltafel nachschlagbare Form[9].

Die bisher aufgeführten Graphen bilden eigentlich nur die Spitze eines Berges von unterschiedlichen Graphen. So können z.B. Graphen auch zur Visualisierung des Informationsgewinns herangezogen werden, indem die Ecken mit sogenannten Informationsvektoren beschriftet werden und die Kanten mit Tests.

Hinter all diesen unterschiedlichen Graphen verbirgt sich für den Informatiker einer der vielen Spagate, die er tagtäglich zu bewältigen hat: auf der einen Seite muß er eine Darstellungsform finden, in der das zu lösende Problem inkl. der dazu benötigten Informationen und Daten adäquat repräsentiert werden kann, andererseits sollte diese Darstellung aus bekannten Repräsentationsformalismen aufgebaut werden, um die dort gewonnen Erkenntnisse verwenden zu können.

1.7 Zusammenhang

In den folgenden Abschnitten werden Begriffe eingeführt, die vor allem im Umfeld des Durchlaufens eines gegebenen Graphens Verwendung finden.

1.7.1 Kantenfolgen, Wege, Kreise

Definition 1.18 (Kantenfolge) *In einem Graphen $G(V, E)$ ist eine* Kantenfolge *eine Folge, deren Glieder abwechselnd Ecken und Kanten sind:*

$$v_0\, e_1\, v_1\, e_2\, v_2\, \ldots\, e_k\, v_k$$

wobei $0 < k \in \mathbb{N}$ und für $i = 1, \ldots, k$ gilt: $s\_t(e_i) = \{v_{i-1}, v_i\}$. Für gerichtete Kanten e_i müssen darüber hinaus $s(e_i) = v_{i-1}$ und $t(e_i) = v_i$ sein. Im Fall $v_0 = v_k$ heißt die Kantenfolge geschlossen.

Falls G keine Mehrfachkanten besitzt, ist die Kantenfolge bereits durch die Folge $v_0, \ldots, v_k$ der Ecken hinreichend beschrieben[10].

[9] vgl. N.J.Nilsson, Principles of AI, Berlin-Heidelberg-New York 1982, S.40 ff

[10] Die Kanten zwischen den einzelnen Ecken sind wegen der Voraussetzung eindeutig bestimmt.

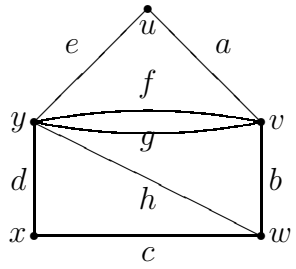
Definition 1.19 (Weg) *Eine Kantenfolge heißt ein* Weg *von v_0 nach v_k, wenn alle Ecken $v_0, \ldots, v_k$ (und damit auch alle Kanten $e_1, \ldots, e_k$) voneinander verschieden sind.*

Die Anzahl der Kanten eines Weges wird als **Länge** *des Weges bezeichnet.*

Definition 1.20 (Kreis) *Eine geschlossene Kantenfolge heißt ein* Kreis*, wenn alle Ecken $v_0, \ldots, v_{k-1}$ und alle Kanten $e_1, \ldots, e_k$ voneinander verschieden sind und $v_0 = v_k$ gilt.*

Statt dem Begriff Kreis wird in der Literatur bei gerichteten Graphen auch oft der Begriff Zyklus verwendet. Dabei spielt gerade im Zusammenhang mit dem Durchlaufen von Graphen die Zyklenfreiheit eine große Rolle, da dadurch z.B. die Terminierung eines Algorithmus sichergestellt werden kann.

Beispiel 1.13 *In dem Graphen*



ist

$$y\, f\, v\, g\, y\, h\, w\, b\, v\, a\, u$$

eine Kantenfolge. Ein Weg in diesem Graphen ist beispielsweise

$$x\, d\, y\, f\, v\, b\, w,$$

und ein Beispiel für einen Kreis ist

$$u\, a\, v\, b\, w\, h\, y\, e\, u.$$

Führen wir in diesen Abschnitten auch vorwiegend nur Begriffe ein, so sind für Informatiker die unterschiedlichen Aussagen über gewisse Charakteristika sehr oft von praktischer Relevanz. So etwa folgender Satz, der letztendlich einen einfachen Algorithmus vorgibt, wie die Zyklenfreiheit eines gerichteten Graphen überprüft werden kann.

Satz 1.5 *Sei $G = (V, E)$ ein gerichteter Graph und es gelte $\forall v \in V : d_+(v) > 0$ oder $\forall v \in V : d_-(v) > 0$, dann besitzt G einen Kreis.*

Der/die LeserInnen mögen sich einfach einen Graphen mit der beschriebenen Voraussetzung aufzeichnen.

Definition 1.21 (Erreichbarkeit) *Eine Ecke u heißt von einer Ecke v aus* erreichbar*, wenn entweder $u = v$ ist oder es eine Kantenfolge gibt, in der v vor u auftritt.*

Die Erreichbarkeit spielt besonders bei gerichteten Graphen eine wichtige Rolle, da hier die Kanten nicht in jeder beliebigen Richtung durchlaufen werden können. Bei ungerichteten Graphen besteht „lediglich“ die Frage nach dem kürzesten Weg zwischen zwei Ecken.

1.7.2 Komponenten

Definition 1.22 (Zusammenhang zweier Ecken)

1. *In einem ungerichteten Graphen heißen zwei Ecken u und v* zusammenhängend*, wenn $u = v$ ist oder es einen Weg von u nach v gibt.*
2. *In einem gerichteten Graphen heißen zwei Ecken u und v* stark zusammenhängend*, wenn $u = v$ ist oder es einen Weg von u nach v gibt und es einen Weg von v nach u gibt.*
3. *In einem gerichteten Graphen heißen zwei Ecken u und v* schwach zusammenhängend*, wenn sie in dem zugrundeliegenden ungerichteten Graphen zusammenhängend sind.*

Definition 1.23 (Zusammenhangskomponente) *In einem ungerichteten Graphen $G(V, E)$ ist die Relation $R \subseteq V \times V$ mit*

$$(u, v) \in R :\Leftrightarrow u \text{ und } v \text{ sind zusammenhängend}$$

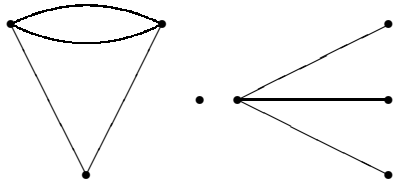
eine Äquivalenzrelation[11] *auf der Eckenmenge V (Begründung?). Die zu den Äquivalenzklassen gehörenden Untergraphen von G heißen* Zusammenhangskomponenten *von G (meist einfach „Komponenten“ genannt). G heißt* zusammenhängend*, falls G genau eine Komponente besitzt.*

Bei gerichteten Graphen werden die Begriffe „Zusammenhangskomponente“ und „zusammenhängend“ im Hinblick auf den zugrundeliegenden ungerichteten Graphen (also im Hinblick auf den schwachen Zusammenhang) verwendet.

[11] Sei U eine Menge. Eine Menge $R \subseteq U \times U$ heißt Äquivalenzrelation auf U, wenn gilt: a) $\forall u \in U : (u, u) \in R$ (Reflexivität), b) $\forall u, v \in U : (u, v) \in R \Rightarrow (v, u) \in R$ (Symmetrie) und c) $\forall u, v, w \in U : (u, v), (v, w) \in R \Rightarrow (u, w) \in R$ (Transitivität).

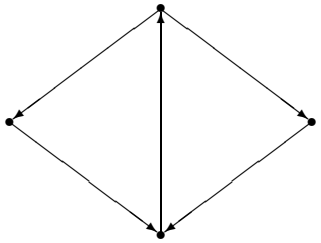
Wenn man ausdrücken will, dass in einer Komponente alle Ecken stark zusammenhängend sind, kann man die Bezeichnungen „starke Zusammenhangskomponente“ bzw. „stark zusammenhängend“ verwenden.

Beispiel 1.14 *1. Dieser ungerichtete Graph besitzt drei Komponenten*[12]*:*



2. Dieser Digraph ist schwach zusammenhängend, aber nicht stark zusammenhängend: (Wieviele starke Zusammenhangskomponenten besitzt er?)

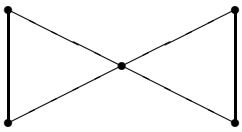
3. Hier ist ein Beispiel für einen stark zusammenhängenden Digraphen:



Bestimmten Ecken kommt u.U. in einem zusammenhängenden Graphen eine besondere Bedeutung zu, nämlich wenn sie die einzigen „Klebestellen“ von zwei zusammenhängenden Teilgraphen sind.

Definition 1.24 (Schnittecken und Schnittkanten) *In einem ungerichteten zusammenhängenden Graphen $G(V, E)$ heißt eine Ecke v eine* Schnittecke*, falls der Untergraph H von G mit Eckenmenge $V \setminus \{v\}$ nicht zusammenhängend ist. Entsprechend heißt eine Kante e eine* Schnittkante*, falls der Teilgraph $F(V, E \setminus \{e\})$ von G nicht zusammenhängend ist.*

Beispiel 1.15 *Dieser Graph besitzt eine Schnittecke, aber keine Schnittkante:*



[12]Beachten Sie, dass intuitiv meist mit einem Graphen ein sogenannter zusammenhängender Graph gemeint ist. Hier zeigt das Beispiel einen diskreten Graphen, der aus drei zusammenhängenden Teilgraphen gebildet wird!

Aufgaben

1. Sind in dem Graphen aus Beispiel 1.13 folgende Kantenfolgen Kreise:
   (a) $y\,f\,v\,g\,y$?
   (b) $y\,f\,v\,f\,y$?

2. Vorgelegt sei eine nicht geschlossene Kantenfolge. Überlegen Sie sich ein Verfahren zur Auswahl einer Teilfolge hieraus, die ein Weg von v_0 nach v_k ist.

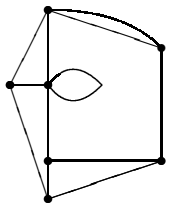
3. (a) Können schlichte (gerichtete? / ungerichtete?) Graphen Kreise mit 2 Kanten besitzen?
   (b) Können bipartite Graphen Kreise mit 3 Kanten besitzen?

4. Definieren Sie den Begriff „starke Zusammenhangskomponente“ analog zu der Begriffsbildung im ungerichteten Fall über eine Äquivalenzrelation.

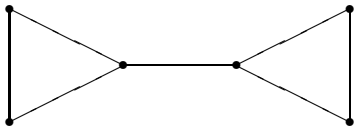
5. Kann es sein, dass ein ungerichteter Graph eine Schnittkante, aber keine Schnittecke besitzt?

6. Wieviele nichtisomorphe zusammenhängende ungerichtete Graphen mit 4 Kanten gibt es? Wieviele davon sind schlicht?

7. Zeigen Sie, dass der nachfolgende Graph zu einem stark zusammenhängenden Digraphen gemacht werden kann, indem Sie allen Kanten geeignete Richtungen geben. Weisen Sie nach, dass Ihr Graph stark zusammenhängend ist!



8. **Zusammenhangskomponente** (? Punkte)
   Prüfen Sie nach, ob der nachfolgende Graph zu einem stark zusammenhängenden Digraphen gemacht werden kann, indem Sie ggf. allen Kanten geeignete Richtungen geben. Weisen Sie nach, dass Ihr Graph stark zusammenhängend ist, oder erklären Sie für den allgemeinen Fall, warum Sie hier keine Lösung finden können!



9. **Zusammenhangskomponente** (? Punkte)
   Eine *starke Zusammenhangskomponente* S eines gerichteten Graphen D ist ein maximaler Untergraph von D, der stark zusammenhängend ist. Dabei bezieht sich „maximal“ darauf, dass S selbst kein Untergraph eines anderen, echten stark zusammenhängenden Untergraphen von D ist.

   Ermitteln Sie alle starken Zusammenhangskomponenten des gerichteten Graphen D in Abbildung 1.2 (Seite 34).

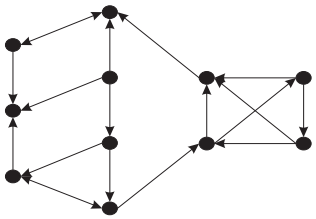


Abbildung 1.2: Ermitteln Sie die starken Zusammenhangskomponenten!

1.8 Speicherung von Graphen

Für Graphen existiert auch eine Beschreibung durch Matrizen oder Listen. Sie ermöglicht beispielsweise eine wenig aufwendige Speicherung von Graphen im Rechner.

Die Motivation, Dinge „anders“ darzustellen ist in der Informatik letztendlich oft darin begründet, sich Arbeit zu ersparen. So werden Graphen durch Matrizen dargestellt, um z.B. durch Charakteristika der Matrizen auf Charakteristika dieser Graphen schließen zu können oder um Algorithmen auf Graphen mittels effizienten Matrizenoperationen realisieren zu können.

Zu beachten ist, dass die Matrizen für sich alleine reine Matrizen sind, d.h. nicht als Graph zu erkennen sind. Diese Festlegung wird bei Realisierungen im Rechner durch die Zugriffsfunktionen vorgenommen. Dort ist festgelegt, wie der Graph in einer Matrix repräsentiert ist (Stichwort: Abstrakter Datentyp). Die im folgenden vorgestellten Formen sind eine von vielen Möglichkeiten, Graphen in einer anderen Notation – hier in Matrizen – darstellen zu können.

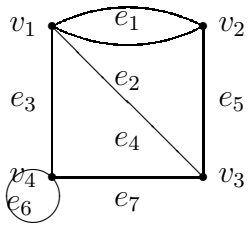
1.8.1 Ungerichtete Graphen

Matrixdarstellung

Definition 1.25 (Adjazenzmatrix) *Die Adjazenzmatrix $A(G) := (a_{ij})$ des ungerichteten Graphen $G(V, E)$ ist eine symmetrische $|V| \times |V|$-Matrix mit:*

$$a_{ij} := \textit{Anzahl der Kanten mit den Endecken } v_i \textit{ und } v_j$$

Beispiel 1.16 *Der Graph $G(V, E)$*



hat nachfolgende Darstellung als Adjazenzmatrix:

$$A(G) = \begin{pmatrix} 0 & 2 & 1 & 1 \\ 2 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

Definition 1.26 (Inzidenzmatrix) *Die Inzidenzmatrix $M(G) := (m_{ij})$ des ungerichteten Graphen $G(V, E)$ ist eine $|V| \times |E|$-Matrix mit:*

$$m_{ij} := \begin{cases} 0, & \textit{falls } v_i \textit{ nicht inzident ist mit } e_j \\ 1, & \textit{falls } v_i \textit{ eine der Endecken von } e_j \textit{ ist} \\ 2, & \textit{falls } v_i \textit{ die Endecke der Schlinge } e_j \textit{ ist} \end{cases}$$

Beispiel 1.17 *Der obige Graph G hat die Inzidenzmatrix*

$$M(G) = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 2 & 1 \end{pmatrix}$$

Welche dieser Darstellungsmöglichkeiten in der Anwendung Verwendung findet, hängt von der Problemstellung ab. U.U. sind beide Formen ungeeignet, da die benötigten Algorithmen auf anderen Darstellungsformen effizienter sind.

Nachbarschaftsliste

Bei Graphen mit niedrigen Eckengraden (d.h. $d(v_i) << |V|$) und ohne Mehrfachkanten enthält die Adjazenzmatrix viele Nullen. In diesem Fall bringt eine Darstellung des Graphen in Form einer Nachbarschaftsliste eine Verringerung des Speicherplatzbedarfs. Eine derartige Liste stellt jeder Ecke ihre Nachbarn gegenüber. Im obigen Beispiel lautet sie:

Ecke		hat gemeinsame Kanten mit
v_1	:	v_2, v_3, v_4
v_2	:	v_1, v_3
v_3	:	v_1, v_2, v_4
v_4	:	v_1, v_3, v_4

Da in diesem Graphen fast alle Ecken miteinander benachbart sind, liefert die Nachbarschaftsliste hier keine wesentliche Verdichtung der Information. Außerdem geht die Information darüber, dass v_1 und v_2 durch mehr als eine Kante verbunden sind hier verloren. Um diese Information zu erhalten müßte vereinbart werden, dass für jede Kante, für die v_i Endecke ist, die andere zugehörige Endecke in die Nachbarschaftsliste aufgenommen wird. In diesem Fall sind dann auch Schleifen darstellbar. Natürlich ist zu beachten, dass bei dieser Form – den Listen – zunächst keine Matrizenoperationen mehr verwendet werden können.

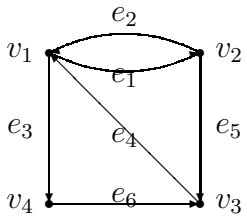
1.8.2 Gerichtete Graphen

Matrixdarstellung

Definition 1.27 (Adjazenzmatrix) *Für einen gerichteten Graphen werden die Einträge der Adjazenzmatrix folgendermaßen definiert:*

$$a_{ij} := \textit{Anzahl der Kanten mit Anfangsecke } v_i \textit{ und Endecke } v_j$$

Beispiel 1.18 *Der gerichtete Graph H*



hat die (nicht symmetrische) Adjazenzmatrix:

$$A(H) = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Definition 1.28 (Inzidenzmatrix) *Für einen schlingenfreien gerichteten Graphen werden die Einträge m_{ij} der Inzidenzmatrix folgendermaßen definiert:*

$$m_{ij} := \begin{cases} 0, & \textit{falls } v_i \textit{ nicht inzident ist mit } e_j \\ -1, & \textit{falls } v_i \textit{ die Anfangsecke von } e_j \textit{ ist} \\ +1, & \textit{falls } v_i \textit{ die Endecke von } e_j \textit{ ist} \end{cases}$$

Auch hier wieder ein Hinweis darauf, dass die gewählte Repräsentationsform in Abhängigkeit der angestrebten „Verarbeitungsziele“ gewählt wird. So könnten in der vorangegangenen Definition durchaus die Zahlen 47, 11, 42 und 88 Verwendung finden. Die gewählte Form erlaubt aber gewisse Aussagen, so z.B. dass die Summe in einer Spalte immer 0 ergeben muß! Auf der anderen Seite sind Schleifen bei obiger Form nicht mehr darstellbar.

Beispiel 1.19 *Der obige Graph H (Seite 36) hat die Inzidenzmatrix*

$$M(H) = \begin{pmatrix} -1 & +1 & -1 & +1 & 0 & 0 \\ +1 & -1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 & +1 & +1 \\ 0 & 0 & +1 & 0 & 0 & -1 \end{pmatrix}$$

Mit Hilfe dieser Matrizen kann man auch bestimme Eigenschaften von Graphen (oder Teilgraphen) „errechnen“. Beispielsweise ist die Beobachtung, dass für einen gerichteten Kreis die Summe der zugehörigen Spalten in der Inzidenzmatrix der Nullvektor ist, Grundlage des folgenden Satzes:

Satz 1.6 *Sei G ein gerichteter Graph mit der Inzidenzmatrix $M(G)$. Der zugrundeliegende ungerichtete Graph werde mit $\tilde{G}$ bezeichnet. Dann gilt:*

1. *Eine Menge von Kanten von G enthält einen Kreis von $\tilde{G}$ genau dann, wenn die zugehörigen Spalten von $M(G)$ linear abhängig sind. Bei der Darstellung des Nullvektors als Linearkombination dieser Spaltenvektoren werden alle Vektoren nur mit -1, 0 oder $+1$ multipliziert.*

2. *Eine Menge von Kanten von G enthält einen Kreis von G genau dann, wenn die zugehörigen Spalten von $M(G)$ linear abhängig sind und bei der Darstellung des Nullvektors als Linearkombination dieser Spaltenvektoren alle Vektoren nur mit 0 oder $+1$ multipliziert werden.*

Beispiel 1.20 *In dem obigen Graphen H bilden z.B. $\{e_1, e_4, e_5\}$ einen Kreis in H, da $e_1 = -1 * e_4 + -1 * e_5$ bzw. $0 = e_1 + e4 + e_5$ gilt. $\{e_2, e_3, e_5, e_6\}$ ist ein Kreis in $\tilde{H}$, da $e_2 = -1 * e_3 + e_5 + -1 * e_6$ bzw. $0 = e_2 + e_3 - e_5 + e_6$ gilt.*

Nachbarschaftsliste

Bei gerichteten Graphen ist die Nachbarschaftsliste genau wie bei ungerichteten aufgebaut. Die einzige zusätzliche Vereinbarung ist, dass die Ecken links die Anfangsecken sind und in den Listen die jeweiligen Endecken der inzidenten Kanten stehen.

Aufgaben

1. Wieviel Speicherplatz benötigt die Adjazenzmatrix eines Graphen? Unterscheiden Sie dabei einerseits zwischen schlichten und nicht schlichten und andererseits zwischen gerichteten und ungerichteten Graphen.

2. Wie können Sie anhand der Adjazenz- oder Inzidenzmatrix eines schlingenfreien Graphen den Grad einer bestimmten Ecke bestimmen?

3. Geben Sie einen effizienten Algorithmus an, der feststellt, ob ein gerichteter, schlichter Graph $G = (V, E)$ in Adjazenzmatrixspeicherung eine Ecke $v \in V$ enthält, mit $d_+(v) = 0$, also eine Ecke, von der kein Pfeil ausgeht, und mit $d_-(v) = |V| - 1$, also eine Ecke, in die $|V| - 1$ Pfeile hineinlaufen. Wieviele solcher Ecken kann dieser Graph haben ? Wieviele Ecken solcher Art könnte er haben, wenn er schlicht wäre ?

4. Sei $A(G)$ die Adjazenzmatrix des (gerichteten oder ungerichteten) Graphen G. Die Einträge a_{ij} lassen sich als Anzahl derjenigen Wege von v_i nach v_j, die aus genau einer Kante bestehen, interpretieren. Geben Sie eine entsprechende Interpretation der Einträge der Matrix $A(G) \cdot A(G)$ an.

5. Wie können Sie der Adjazenzmatrix eines Graphen G ansehen, dass G bipartit ist?

6. Wie erhalten Sie aus der Adjazenzmatrix eines Graphen G die Adjazenzmatrix eines Teil- oder Untergraphen H?

7. Sei $G = (V, E)$ ein gerichteter Graph, $s, t : E \to V$ zwei Funktionen, die jeweils die Anfangsecke (source) und Endecke (target) einer Kante angeben. Der inverse Graph G^{-1} ist definiert durch $G^{-1} = (V, E^{-1})$ mit $E^{-1} := \{e^{-1} | s(e^{-1}) = t(e), t(e^{-1}) = s(e)\}$. Der Graph G^{-1} entsteht aus G also dadrurch, daß die Richtung aller Pfeile umgedreht wird.

   Geben Sie sowohl für die Adjazenzmatrixspeicherung als auch für die Adjazenslistenspeicherung von G effiziente Algorithmen an, um G^{-1} aus G zu berechnen. Schätzen Sie die Laufzeit Ihrer Algorithmen ab!

Kapitel 2

Optimale Wege

2.1 „Breadth First Search“-Technik (BFS)

Vorab folgende Bemerkung zu der Namensgebung: Je nach Kontext wird z.B. die englischsprachige Variante genommen, hier etwa „Breadth First Search“ bzw. „Depth First Search“, oder die deutschsprachige Variante, hier etwa „Breitensuche“ bzw. „Tiefensuche“. Aus Sicht des Konzeptes kann dies dann mit den Speichermedien „First In First Out“ (FIFO, Queue, Schlange) und „Last In First Out“ (LIFO, Stack, Stapel) assoziiert werden, was sich ebenfalls in der Namensgebung äußern kann.

Hier wird nun eine Methode zum Auffinden eines Weges von einer Ecke `s` zu einer anderen Ecke `t` vorgestellt, bei der die kleinstmögliche Anzahl an Kanten einbezogen wird. Ein derartiger Weg – sollte er existieren – wird als *kürzester Weg* von `s` nach `t` bezeichnet, wobei den Ecken in dem Graphen G die Kennzeichnungen 0, 1, 2, ... zugeordnet werden.

Algorithmus 2.1 *Gegeben sei ein Graph G mit zwei ausgezeichneten Ecken* `s` *und* `t`.

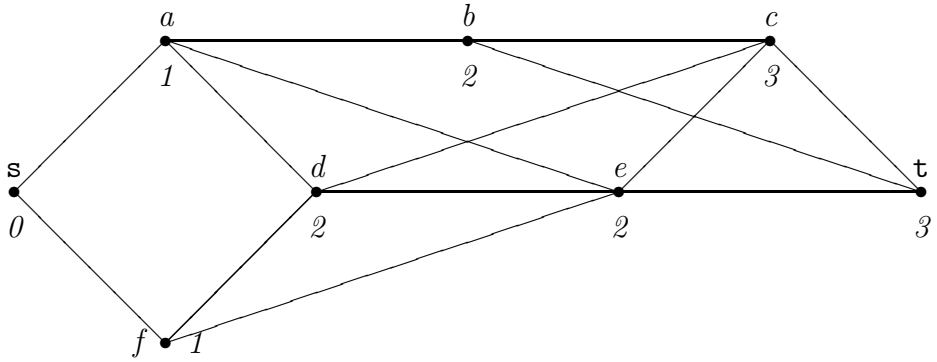
Schritt 1: *Man kennzeichne die Ecke* `s` *mit 0 und setze $i = 0$.*

Schritt 2: *Man ermittle alle nichtgekennzeichneten Ecken in G, die zu dem mit i gekennzeichneten Ecken benachbart sind.*
Falls es derartige Ecken nicht gibt, ist `t` *nicht mit* `s` *über einen Weg verbunden.*
Falls es derartige Ecken gibt, sind sie mit $i + 1$ zu kennzeichnen.

Schritt 3: *Wenn* `t` *gekennzeichnet wurde, folgt Schritt 4, wenn nicht, erhöhe man i um eins und gehe zu Schritt 2.*

Schritt 4: *Die Länge des kürzesten Weges von* `s` *nach* `t` *ist* $i+1$. *Der Algorithmus wird beendet.*

Beispiel 2.1 *Im Falle des folgenden Graphen geht man z.B. so vor, daß zuerst* `s` *mit 0, danach* `a, f` *mit 1 und dann* `b, d, e` *mit 2 gekennzeichnet werden. Anschließend wird* `c, t` *folgerichtig die 3 zugeordnet. Da* `t` *mit 3 gekennzeichnet ist, ist 3 die Länge des kürzesten Weges von* `s` *nach* `t`.



Aufgrund der durch den BFS-Algorithmus erzeugten Kennzeichnung der Ecken kann ein kürzester Weg von s nach t nun explizit angegeben werden. Die Kennzeichnung der Ecke `a` wird dabei mit $\lambda(a)$ bezeichnet.

Algorithmus 2.2 *Verwendung der Kennzeichnung, die durch den BFS-Algorithmus erzeugt wurde. Der rückverfolgende Algorithmus erzeugt einen Weg* $v_0, v_1, \ldots, v_{\lambda(t)}$, *so daß* $v_0 = s; v_{\lambda(t)} = t$ *ist.*

Schritt 1: *Man setze* $i = \lambda(t)$ *und ordne* $v_i = t$ *zu.*

Schritt 2: *Man ermittle eine Ecke* u, *der zu* v_i *benachbart ist und mit* $\lambda(u) = i-1$ *gekennzeichnet ist. Man ordne* $v_{i-1} = u$ *zu.*

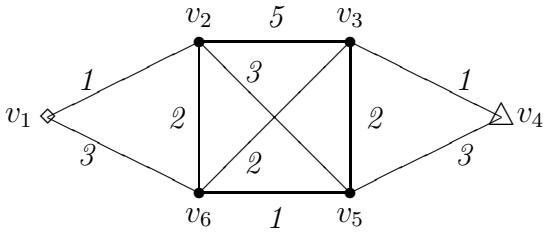
Schritt 3: *Wenn* $i = 1$ *ist, ist der Algorithmus beendet. Wenn nicht, erniedrige man* i *um eins und gehe zu Schritt 2.*

Beispiel 2.2 *(Fortsetzung) Bei dem Graphen aus dem Beispiel 2.1 haben wir z.B.* $\lambda(t) = 3$, *so daß wir mit* $i = 3$ *starten und* $v_3 = t$ *ist. Wir können dann eine zu* $v_3 = t$ *benachbarte Ecke wählen, etwa* e *mit* $\lambda(e) = 2$ *(*b *wäre auch möglich gewesen), und ordnen* $v_2 = e$ *zu. Als nächstes können wir dann eine zu* $v_2 = e$ *benachbarte Ecke wählen, etwa* f *mit* $\lambda(f) = 1$ *(*a *wäre auch möglich gewesen), und ordnen* $v_1 = f$ *zu. Schließlich nehmen wir das zu* f *benachbarte* s *mit* $\lambda(s) = 0$ *und ordnen* $v_0 = s$ *zu. Dies ergibt den kürzesten Weg* $v_0v_1v_2v_3 = sfet$ *von* s *nach* t.

2.2 Das Problem der kürzesten Wege

Wir erweitern die Menge der durch einen Graphen darstellbaren Sachverhalte, indem wir in diesem Kapitel jeder Kante eine reelle Zahl zuordnen, die wir als „Länge“ dieser Kante bezeichnen. Man spricht dann von einem Graphen mit bewerteten Kanten. In der zeichnerischen Darstellung eines Graphen notieren wir die Kantenbewertung unmittelbar neben der entsprechenden Kante.

Beispiel 2.3 *Sechs Städte werden durch das folgende Eisenbahnnetz miteinander verbunden. Die Länge einer Kante gibt dabei die durchschnittlich zu erwartende Fahrzeit in Stunden an.*



Für einen Reisenden, der von ⋄-Stadt nach △-City unterwegs ist, stellt sich die Frage, auf welchem Weg er am schnellsten zum Ziel kommt.

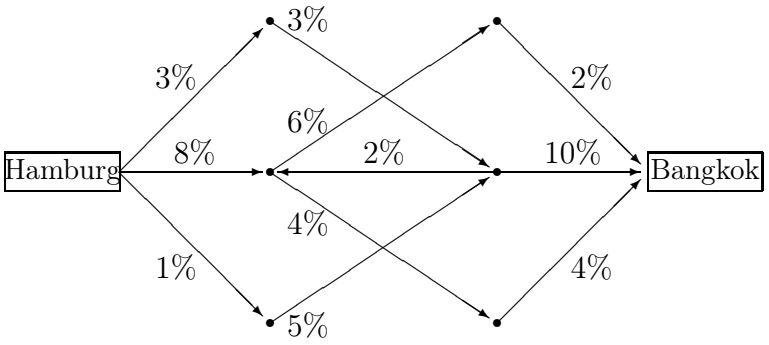
Allgemein verstehen wir unter einem Problem des kürzesten Weges die folgende Aufgabenstellung:
In einem (gerichteten oder ungerichteten) zusammenhängenden Graphen $G(V, E)$ mit bewerteten Kanten wird ein Weg (ggf. ein gerichteter Weg) von einer gegebenen Ecke v_1 zu einer gegebenen Ecke v_n gesucht, der von allen derartigen Wegen die minimale Kantenbewertungssumme besitzt.

Im folgenden Abschnitt wird ein Algorithmus zur Lösung dieser Aufgabe vorgestellt. Tatsächlich leistet er jedoch mehr als in der Aufgabenstellung verlangt. Wir können daher die Formulierung der Aufgabe erweitern:
In einem (gerichteten oder ungerichteten) zusammenhängenden Graphen $G(V, E)$ mit bewerteten Kanten wird je ein (ggf. gerichteter) Weg mit minimaler Kantengewichtssumme von einer gegebenen Ecke v_1 zu jeder anderen Ecke v_i, $i = 2, \ldots, |V|$ gesucht.

Aufgaben

1. Machen Sie einen Vorschlag zur Speicherung von Graphen mit bewerteten Kanten. Setzen Sie dabei voraus, dass diese Graphen schlicht sind (ggf. löschen Sie bei Mehrfachkanten alle bis auf diejenige mit der niedrigsten Bewertung).

2. Eine Firma in Hamburg will einer Firma in Bangkok einen Geldbetrag überweisen. Für die Übermittlung des Geldes stehen verschiedene Wege zur Verfügung. Leider vermindert sich die Summe auf jedem Teilstück des Übertragungsweges um einen gewissen Prozentsatz (Bearbeitungsprovision) des jeweils verbliebenen Betrages:



Für die Abwicklung der Überweisung soll ein Weg gewählt werden, auf dem beim Empfänger ein möglichst hoher Anteil der ursprünglich angewiesenen Summe ankommt.

(a) Ändern Sie die Kantenbewertungen zunächst so, dass die Lösung der Aufgabe ein Weg von Hamburg nach Bangkok mit *maximalem Produkt* der Kantenbewertungen ist.

(b) Formen Sie die Kantenbewertungen anschließend so um, dass durch die Aufgabenstellung ein Weg mit *minimaler* Kantengewichts*summe* gesucht wird. (Tip: Übergang zum Logarithmus)

2.3 Der Algorithmus von Dijkstra

2.3.1 Grundidee

Der Algorithmus von Dijkstra zur Bestimmung kürzester Wege von einer fest vorgegebenen Ecke (im folgenden v_1 genannt) zu allen anderen Ecken eines schlichten zusammenhängenden gerichteten Graphen $G(V, E)$ mit endlicher Eckenmenge und nichtnegativ bewerteten Kanten ist ein iteratives Verfahren, d.h. eine bestimmte Folge von Arbeitsanweisungen wird mehrmals wiederholt. Bei jeder Wiederholung dieser Folge wird dabei für genau eine neue Ecke v_i festgestellt, welche Kantenbewertungssumme ein Weg von v_1 nach v_i mindestens hat, und es wird ein Weg mit einer solchen minimalen Gesamtlänge angegeben.

Bei der Anwendung dieses Verfahrens auf ungerichtete Graphen muß jede Kante durch ein Paar entgegengesetzt gerichteter Kanten ersetzt werden.

2.3.2 Verfahrensvorschrift

Algorithmus 2.3 *Der Algorithmus besteht aus einer Vorbereitungs- und einer Iterationsphase.*

Vorbereitung *Es bezeichne l_{ij} die Länge der Kante v_iv_j. Falls es keine Kante v_iv_j gibt, sei $l_{ij} := \infty$ Für jede Ecke $v_i \in V$ des zu untersuchenden Graphen werden drei Variable angelegt:*

1. *$Entf_i$ gibt die bisher festgestellte kürzeste Entfernung von v_1 nach v_i an. Der Startwert ist 0 für $i = 1$ und ∞ sonst.*
2. *$Vorg_i$ gibt den Vorgänger von v_i auf dem bisher kürzesten Weg von v_1 nach v_i an. Der Startwert ist v_1 für $i = 1$ und undefiniert sonst.*
3. *OK_i gibt an, ob die kürzeste Entfernung von v_1 nach v_i schon bekannt ist. Der Startwert für alle Werte von i ist false.*

Iteration *Wiederhole (i, j seien dabei die Laufvariablen, h ein fester Wert)*

- *Suche unter den Ecken v_i mit $OK_i = false$ eine Ecke v_h mit dem kleinsten Wert von $Entf_i$.*
- *Setze $OK_h := true$. (Wieso?)*
- *Für alle Ecken v_j mit $OK_j = false$, für die die Kante v_hv_j existiert:*
  - *Falls gilt $Entf_j > Entf_h + l_{hj}$ dann*
    - *Setze $Entf_j := Entf_h + l_{hj}$*
    - *Setze $Vorg_j := h$*

solange es noch Ecken v_i mit $OK_i = false$ gibt.

Am Ende des Algorithmus steht in $Entf_i$ die Länge des kürzesten Weges von v_1 nach v_i. Der Weg selbst wird von v_i aus durch Rückwärtsverfolgen der Angaben in dem Vektor *Vorg* gefunden.

Die Korrektheit des Algorithmus wird durch die Gültigkeit folgender Schleifeninvariante sichergestellt: Vor und nach jedem Durchlauf durch die „Wiederhole“-Schleife enthält jedes $Entf_j$ die Länge eines kürzesten Weges von v_1 nach v_j unter der Zusatzbedingung, dass unterwegs nur Ecken v_k mit $OK_k = true$ passiert werden dürfen.

Beispiel 2.4 *In dem obigen Eisenbahnnetz (Seite 43) sollen von $v_1 :=$ $\diamond$-Stadt aus die kürzesten Verbindungen zu allen anderen Städten bestimmt werden. Nach der Vorbereitungsphase haben Entf, Vorg und OK die Werte:*

i	1	2	3	4	5	6
Entf	0	∞	∞	∞	∞	∞
Vorg	1					
OK	*f*	*f*	*f*	*f*	*f*	*f*

Hier ein Zwischenschritt nach drei Iterationsschritten ($v_h = v_6$):

i	1	2	3	4	5	6
Entf	0	1	5	∞	4	3
Vorg	1	1	6		2	1
OK	*t*	*t*	*f*	*f*	*f*	*t*

Nach der Auswahl von v_2 wurde der Vorgänger von v_6 nicht verändert, da die Entfernung nicht kleiner, sondern nur gleich war.

Der Algorithmus macht übrigens keine Angaben darüber, was zu tun ist, wenn mehr als eine Ecke die Auswahlbedingung erfüllt: dann besteht freie Wahl!

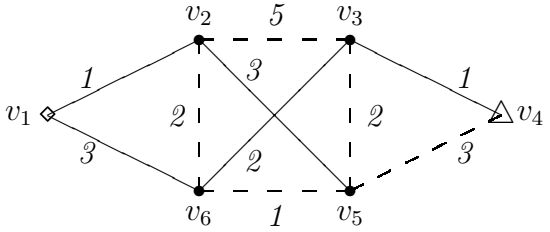
In den Iterationsschritten ist v_h nacheinander die Ecke v_1, v_2, v_6, v_5, v_3, v_4, und am Ende haben Entf, Vorg und OK die Werte:

i	1	2	3	4	5	6
Entf	0	1	5	6	4	3
Vorg	1	1	6	3	2	1
OK	*t*	*t*	*t*	*t*	*t*	*t*

Die Ecke v_4 hat also von v_1 den Abstand 6, und auf dem Weg von v_1 nach v_4 hat die vorletzte Ecke den Index $Vorg_4 = 3$ sowie die drittletzte Ecke den Index $Vorg_3 = 6$ und wegen $Vorg_6 = 1$ lautet dann der komplette Weg:

$$v_1,\ v_6,\ v_3,\ v_4$$

In der folgenden Darstellung sind die nicht zu den kürzesten Wegen gehörenden Kanten gestrichelt dargestellt:



Es fällt in dem Beispiel auf, dass es zu manchen Ecken von v_1 aus mehrere gleichlange kürzeste Wege gibt; z.B. hat v_1, v_2, v_6 dieselbe Länge wie die Kante v_1v_6. Welcher dieser Wege vom Algorithmus dann tatsächlich gefunden wird, hängt davon ab, in welcher Reihenfolge die Ecken als v_h auftreten. Da der Wert von $Entf_i$ nur umgesetzt wird, wenn ein echt kürzerer Weg von v_1 nach v_i gefunden wird, schließt die zuerst gefundene kürzeste Verbindung alle später auftretenden ebensolangen Wege aus.

2.3.3 Arbeitsaufwand

Es ist offensichtlich, dass der Dijkstra-Algorithmus in endlicher Zeit zu einem Ergebnis kommt. Da der vorgelegte Graph nur endlich viele Ecken und Kanten besitzt, könnte man dasselbe aber auch von zahlreichen anderen wesentlich umständlicheren Vorgehensweisen sagen. Es ist deshalb nützlich, einen Bewertungsmaßstab zur Verfügung zu haben, der eine genauere Aussage über die Leistungsfähigkeit eines Algorithmus ermöglicht.

Komplexität

Ein solcher Maßstab ist z.B. die Zeitkomplexität im schlimmsten Fall (worst case) – meist einfach Komplexität genannt. Sie macht eine Aussage über die Chrakteristik der Größenordnung der Laufzeit des Algorithmus in Abhängigkeit vom Umfang der Eingabedaten.

Beim Dijkstra-Algorithmus haben wir folgende Situation: Als Eingabedaten werden $|V| \cdot (|V| - 1)$ Kantenlängen benötigt (falls der Graph ungerichtet ist, nur halb so viele). Angaben darüber, zu welcher Kante eine gewisse Kantenlänge gehört, sind entbehrlich, wenn man zuvor eine feste Reihenfolge (z.B. lexikographisch aufsteigend) festlegt. Die ausdrückliche Angabe, ob eine gewisse Kante in dem Graphen überhaupt existiert, läßt sich dadurch umgehen, dass für nicht existierende Kanten eine besonders große Kantenlänge (z.B. $|V|$-mal die Länge der längsten existierenden Kante) angegeben wird.

Die Anzahl der bei der Ausführung des Algorithmus maximal benötigten Arbeitsschritte lautet:
– zur Vorbereitung müssen $3 \cdot |V|$ Variable initialisiert werden;
– bei der Iteration muß $(|V| - 1)$-mal wiederholt werden: suche eine von höchstens $|V| - 1$ Ecken und inspiziere ihre höchstens $|V| - 1$ Nachbarn.
Zusammen sind dies höchstens $3 \cdot |V| + 2 \cdot (|V| - 1)^2$ Arbeitsschritte.

Diese Angaben sind allerdings für die Beurteilung der Komplexität ungebräuchlich. Dies hängt einmal damit zusammen, dass diese Ausdrücke je nach Algorithmus recht unübersichtlich werden können, zum anderen werfen

sie Fragen auf, die nichts mit dem Algorithmus, sondern vielmehr mit dem verwendeten Rechner zu tun haben (In welchem Zahlensystem wird eine Kantenlänge gespeichert? Wieviel Zeit vergeht während der Ausführung eines Arbeitsschritts?).

Deshalb interessiert man sich letztlich nicht für die genaue Form der obigen Ausdrücke, sondern nur für ihre Größenordnung bzw. ihre Charakteristik.

Zwei Funktionen $f(x)$ und $g(x)$ haben (intuitiv) dasselbe Wachstumsverhalten, falls es Konstanten c und d gibt, so dass für alle genügend große x gilt:

$$d < \frac{f(x)}{g(x)} < c \quad \text{und} \quad d < \frac{g(x)}{f(x)} < c$$

Neben der Laufzeit ist auch der Speicherbedarf zu berücksichtigen. Komplexität wird aber ausführlich im Bereich „Algorithmen und Datenstrukturen“ untersucht. Die drei wesentlichen asymptotischen Notationen sind:

- $O(g(x)) := \{f(x) | \exists c > 0 \exists x_0 > 0 \forall x > x_0 : f(x) \leq c * g(x)\}$
- $\Omega(h(x)) := \{f(x) | \exists c > 0 \exists x_0 > 0 \forall x > x_0 : f(x) \geq c * h(x)\}$
- $\Theta(k(x)) := \{f(x) | \exists c_1, c_2 > 0 \exists x_0 > 0 \forall x > x_0 : c_1 * k(x) \leq f(x) \leq c_2 * k(x)\}$

Da wir hier über Zeit-/Speicherschranken reden, sind die Funktionen alle positiv, da wir ja in dem Sinne Zeit bzw. Speicher verbrauchen. Bei der Funktion $f(x)$ ist zu beachten, dass hier ein genauer Verlauf nicht immer gut bestimmbar ist: so wird z.B. für die O-Notation die „worst case“-Analyse betrieben und für die Ω-Notation die „best case“-Analyse.

Definition 2.1 *Eine reelle Funktion $f(x)$ heißt* von der Größenordnung $g(x)$ *(geschrieben: $f(x) = O(g(x))$[1]), wenn $g(x)$ das Verhalten von $f(x)$ für hinreichend große x im wesentlichen bestimmt – präziser:*

$$f(x) = O(g(x)) \Longleftrightarrow \exists c \in \mathbb{R} : \lim_{x \to \infty} \frac{f(x)}{g(x)} = c$$

Beispiel 2.5

$$\begin{aligned} \frac{6}{5}x^2 - 5x &= O(x^2) \\ -x^5 + \ln x &= O(x^5) \\ 2^x - 5x^3 &= O(2^x) \end{aligned}$$

[1]Die Schreibweise = hat sich hier durchgesetzt. Korrekt müsste es $f(x) \in O(g(x))$ heißen

Für den Dijkstra-Algorithmus heißt das: Eingabeumfang und Arbeitsaufwand haben größenordnungsmäßig die Werte

$$\begin{array}{rcl} \text{Umfang der Eingabedaten} & : & O(|V|^2) \\ \text{Anzahl der Arbeitsschritte} & : & O(|V|^2) \end{array}$$

Der Arbeitsaufwand ist also größenordnungsmäßig gleich dem Eingabeumfang.[2]

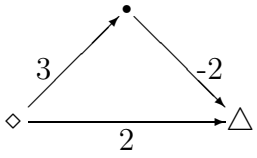
Definition 2.2 (Polynomialer Algorithmus) *Ein Algorithmus, bei dem die Größenordnung des Arbeitsaufwands durch eine Potenz der Größenordnung des Eingabeumfangs beschränkt ist, heißt* polynomial. *Wenn es keine solche Schranke gibt, heißt der Algorithmus* exponentiell.

Die Klassifizierung eines Algorithmus als polynomial hat sich aus dem Grunde als ein praktikables Kriterium erwiesen, weil zum einen Polynome bezüglich der Komposition von Funktionen abgeschlossen sind, also die Verwendung eines polynomialen Algorithmus als Unterprogramm nichts daran ändert, ob das Hauptprogramm polynomial ist oder nicht (was für die Frage nach der Zugehörigkeit zu einer echten Teilmenge der polynomialen Algorithmen nicht gilt) und weil eine Erhöhung der Rechengeschwindigkeit der verwendeten Hardware um einen bestimmten Prozentsatz den in einer festen Zeit bewältigbaren Eingabeumfang bei einem polynomialen Algorithmus ebenfalls um einen festen Prozentsatz beschleunigt (während bei exponentiellen Algorithmen die Auswirkungen einer Verbesserung der Hardware mit zunehmender Problemgröße „verpuffen“).

2.3.4 Versagen des Algorithmus bei negativen Kantenlängen

Bei der Vorstellung des Dijkstra-Algorithmus haben wir vorausgesetzt, dass alle Kantenbewertungen nichtnegativ sind. Bereits einzelne negativ bewertete Kanten können den Dijkstra-Algorithmus zu einem falschen Ergebnis bringen, wie das folgende Beispiel zeigt:

[2] Eigentlich müßte man an dieser Stelle auch noch den Speicherplatzbedarf jedes Eingabedatums l_{ij} und den Rechenzeitbedarf jedes Arbeitsschritts (Addition, Wertzuweisung, Vergleich) in die Betrachtung einbeziehen. Aus folgendem Grund ändert dies aber nichts an dem angegebenen Resultat: Wenn wir unterstellen, dass alle Kantenlängen ganzzahlig sind und in einem Stellenwertsystem dargestellt werden und l_{max} die größte dieser Zahlen ist, dann erhalten wir für Eingabeumfang und Arbeitsaufwand jeweils $O(|V|^2 \cdot \log(l_{max}))$. In Abschnitt 4.1.4 werden wir jedoch eine Situation kennenlernen, bei der derartige Überlegungen erforderlich sind.



Der Dijkstra-Algorithmus liefert als Wert des kürzesten Weges von $\diamond$ nach $\triangle$ den Wert 2. 1 (= 3 -2) wäre jedoch das korrekte Ergebnis!

Aufgaben

1. Lösen Sie das Überweisungsproblem aus dem vorigen Abschnitt mit Hilfe des Dijkstra-Algorithmus.

2. Bestimmen Sie in dem Eisenbahnnetz kürzeste Wege zwischen $\triangle$-City und allen anderen Orten.

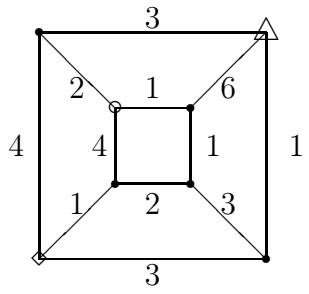
3. Warum läßt sich der Dijkstra-Algorithmus in dem Fall, dass einige Kanten eine negative Länge besitzen nicht auf die folgende Weise „retten“?

   (a) Addiere zu jeder Kantenlänge dieselbe positive Zahl, so dass alle Kantenlängen positiv werden,

   (b) wende den Dijkstra-Algorithmus an, und

   (c) subtrahiere von allen Kantenlängen wieder die unter a) addierte Zahl.

4. Bestimmen Sie in dem folgenden Graphen mit Hilfe des Dijkstra-Algorithmus den kürzesten Weg von $\diamond$ über $\circ$ nach $\triangle$:

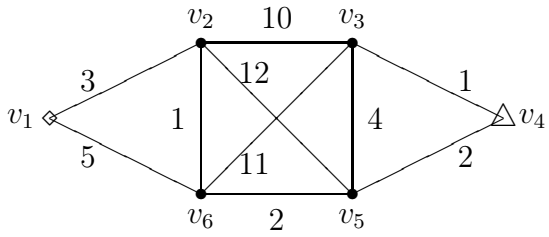


5. Zur Lösung eines Problems stehen zwei Algorithmen zur Verfügung. Bei einer Größenordnung des Eingabeumfangs von L hat der eine Algorithmus eine Komplexität von $O(L^3)$ und der andere eine Komplexität von $O(2^L)$. Im Fall von 100 Eingabedaten benötigen beide Algorithmen auf einem bestimmten Rechner 1 Stunde Rechenzeit. Wieviele Eingabedaten

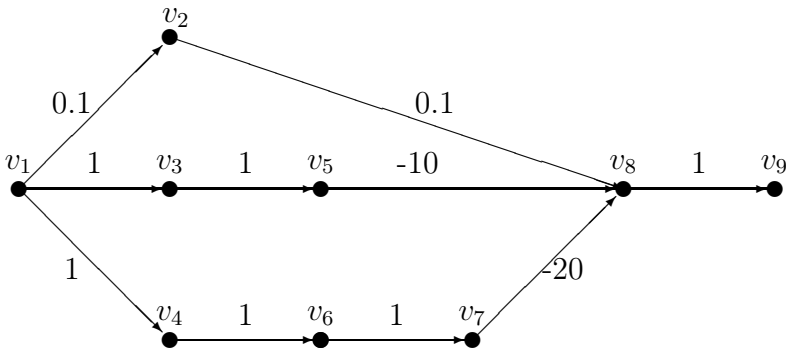
dürfen Probleme haben, damit sie mit Hilfe eines der Algorithmen jeweils in 1 Stunde gelöst werden können, wenn der Rechner durch sein achtmal so schnelles Nachfolgemodell ersetzt wird?

6. Begründen Sie, weshalb bei der Anwendung des Dijkstra-Algorithmus auf einen ungerichteten zusammenhängenden Graphen mit nichtnegativen Kantenlängen das entstehende System kürzester Wege ein Baum ist.

7. Führen Sie den Algorithmus von Dijkstra mit nachfolgendem Graphen durch (bis zum Abbruch!). Wählen Sie als Aussgangspunkt die Ecke v_1. Fertigen Sie eine Dokumentation an, aus der der Ablauf deutlich wird.



8. **Algorithmus von Dijkstra** (? Punkte)
Führen Sie den Algorithmus von Dijkstra mit nachfolgendem Graphen bis zum Abbruch durch!. Wählen Sie als Aussgangspunkt die Ecke v_1. Fertigen Sie eine Dokumentation an, aus der der Ablauf deutlich wird. Zeigen Sie, dass der Algorithmus nicht den kürzesten Weg von v_1 nach v_9 findet und begründen Sie in allgemeiner Form, woran dies liegt!



9. **Algorithmus von Dijkstra** (? Punkte)
Führen Sie den Algorithmus von Dijkstra mit dem Graphen in Abbildung 2.1 (Seite 52) durch (bis zum Abbruch!). Wählen Sie als Aussgangspunkt

die Ecke x_7. Fertigen Sie eine Dokumentation an, aus der der Ablauf deutlich wird. Bei **Wahlmöglichkeiten** ist immer die lexikographisch niedrigste Ecke zu wählen, bzw. die mit dem kleinsten Index.

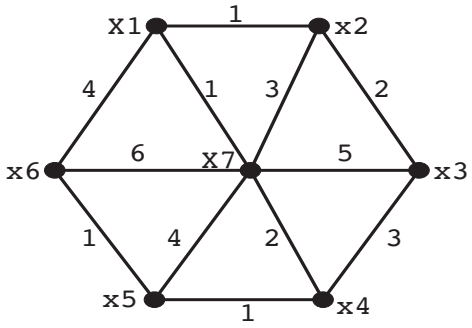


Abbildung 2.1: Optimaler Weg gesucht!

10. **Algorithmus von Dijkstra** (? Punkte)

    (a) Führen Sie den Algorithmus von Dijkstra mit dem Graphen in Abbildung 2.2 (Seite 53) durch (bis zum Abbruch!). Wählen Sie als Aussgangspunkt/Startecke die Ecke X_5. Fertigen Sie eine Dokumentation an, aus der der Ablauf deutlich wird. Bei **Wahlmöglichkeiten** ist immer die Ecke mit dem kleinsten Index zu wählen.

    Geben Sie dann den teuersten Weg an und den bzgl. Anzahl der Kanten längsten Weg. Bei Wahlmöglichkeiten wählen Sie bei den Kosten den bzgl. Kanten kürzesten Weg und bei der Länge den bzgl. Kosten günstigsten Weg aus.

    (b) Erläutern Sie, was man allgemein unter einem Greedy-Algorithmus versteht und begründen Sie, dass der Algorithmus von Dijkstra zu dieser Gruppe von Algorithmen gehört.

    Nennen Sie Problembeispiele außerhalb der Graphentheorie, bei denen der Greedy-Ansatz zum raschen Erfolg führen könnte.

    (c) Nimmt man den Algorithmus von Dijkstra als Grundlage, so lässt sich mit minimalen Änderungen ein Algorithmus erstellen, welcher einen minimalen Spannbaum bzw. ein minimales Gerüst im Graphen berechnet.

        i. Welche Änderungen müssen lediglich vorgenommen werden ? Bitte notieren Sie nur die Änderungen, nicht den ganzen Algorithmus.

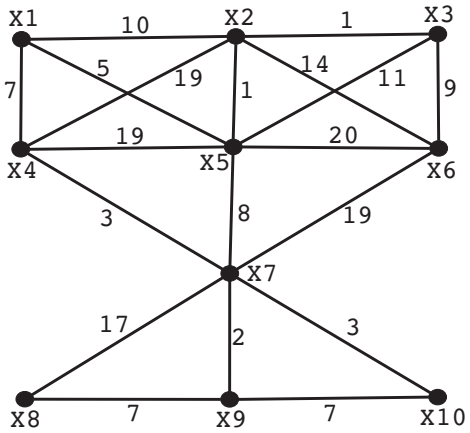


Abbildung 2.2: Optimaler Weg gesucht!

ii. Die Bedeutung der Variablen `Entf` und `Vorg` ist nun eine andere als beim Dijkstra-Algorithmus. Erläutern Sie diese im Unterschied zur ursprünglichen Bedeutung.

2.4 Weitere Kürzeste-Wege-Algorithmen

Neben dem Dijkstra-Algorithmus existieren weitere Verfahren zum Auffinden kürzester Wege. Beispielhaft sollen hier zwei Abwandlungen des Dijkstra-Algorithmus und ein Verfahren, das auf einem gänzlich anderen Ansatz basiert, vorgestellt werden.

Der Unterschied besteht im Wesentlichen darin:

Dijkstra-Algorithmus Sucht den optimalen Weg von einer Ecke zu allen anderen Ecken. Negative Kosten (z.B. Energiegewinn beim Berg abwärtsfahren mit einem Elektromotor) werden verarbeitet, jedoch liefert der Algorithmus dann falsche Ergebnisse. Die Auswahl der zu untersuchenden nächsten Ecke in Kombination mit der Variablen OK_i setzen rein positive Kosten voraus. Bzgl. der Iteration läuft der Algorithmus $(|V|-1)\cdot$ maximaler Eckengrad mal.

Bellmann-Ford Algorithmus Sucht den optimalen Weg von einer Ecke zu allen anderen Ecken. Negative Kosten können verarbeitet werden. Sollten diese zu einem negativen Kreis führen (also $-\infty$ als optimale Kosten), so wird dies erkannt. Bzgl. der Iteration läuft der Algorithmus $(|V|-1)\cdot|E|$ mal.

FiFo Algorithmus Sucht den optimalen Weg von einer Ecke zu allen anderen Ecken. Negative Kosten können verarbeitet werden. Sollten diese zu einem negativen Kreis führen (also $-\infty$ als optimale Kosten), so wird dies nicht erkannt (die Queue wird niemals leer sein). Bzgl. der Iteration läuft der Algorithmus $(|V|-1)\cdot$ Änderungen mal.

Dieser Algorithmus ist gegenüber dem Bellmann-Ford günstiger, wenn der Graph relativ gleiche Kosten hat, da er sich durch die Änderungen triggern lässt. Bei einem in den Kosten sehr unterschiedlichem Graphen ist der Bellmann-Ford günstiger, da er in einem Schritt mehr als eine Änderung durchführen kann, ohne dass dies zu einer erhöhten Anzahl an Iterationen führt.

Floyd-Warshall Algorithmus Sucht den optimalen Weg von allen Ecken zu allen anderen Ecken. Negative Kosten können verarbeitet werden. Sollten diese zu einem negativen Kreis führen (also $-\infty$ als optimale Kosten), so wird dies erkannt. Bzgl. der Iteration läuft der Algorithmus $|V|^3$ mal.

Dieser Algorithmus ist gegenüber den anderen günstiger, wenn der Graph relativ lange stabile Kosten hat, da er dann zu Beginn berechnet werden kann und lange als Informationsquelle dienen kann. Bei dynamischen Kosten sind die anderen Algorithmen günstiger, da sie bei Bedarf auf den aktuellen Kosten effizienter die Berechnung durchführen.

2.4.1 Der Algorithmus von Bellmann-Ford

Der Algorithmus von Bellmann-Ford kann als eine Verallgemeinerung des Algorithmus von Dijkstra verstanden werden. Auch hier wird gesucht, indem von einer ECke aus alle benachbarten Ecken betrachtet werden, jedoch werden in jedem Durchgang stets alle Kanten betrachtet, also in dem Sinne alle Ecken. Im Unterschied zu Dijkstra werden die Ecken zu keinem Zeitpunkt als abgeschlossen betrachtet. Die kürzeste Entfernung einer Ecke zur Ausgangsecke steht also erst dann fest, wenn der Algorithmus endet. Falls eine Ecke vom der Ausgangsecke aus nicht erreichbar ist, wird der Abstand formal als unendlich gesetzt. Dieser Algorithmus kann negative Kosten verarbeiten und erkennt negabtive Kreise.

Algorithmus 2.4 Vorbereitung *l_{ij}: Länge der Kante v_iv_j. $l_{ij} := \infty$, falls es eine solche Kante nicht gibt.*
Für jede Ecke $v_i \in V$ werden zwei Variablen angelegt:

1. *$Entf_i$: die bisher kürzeste Entfernung von v_1 nach v_i an. Startwert 0 für $i = 1$ und ∞ sonst.*

2. *$Vorg_i$ Vorgänger von v_i auf dem bisher kürzesten Weg von v_1 nach v_i an. Startwert v_1 für $i = 1$ und undefiniert sonst.*

Iteration *Wiederhole $|V| - 1$ mal*

- *Für alle Kanten $(v_i v_j) \in E$*
  - *Falls gilt $Entf_j > (Entf_i + l_{ij})$ dann*
    - *Setze $Entf_j := (Entf_i + l_{ij})$*
    - *Setze $Vorg_j := v_i$*

Iteration *Für alle Kanten $(v_i v_j) \in E$*

- *Falls gilt $Entf_j > (Entf_i + l_{ij})$ dann STOP mit Ausgabe „Zyklus negativer Länge gefunden“*

Hinweis: Da über alle Kanten iteriert wird, ist bei einem ungerichteten Graphen darauf zu achten, dass die Kanten in beide Richtungen untersucht werden!

2.4.2 Der FIFO-Algorithmus

Diesen Algorithmus erhält man durch folgende Abwandlung des Dijkstra-Verfahrens:

Die Variable *OK* entfällt. Es wird eine Warteschlange für die Ecken eingeführt. Zu Beginn des Algorithmus enthält sie die Ecke v_1. Wenn $Entf_i$ herabgesetzt wird und v_i nicht in der Schlange enthalten ist, wird die Ecke v_i hinten an die Warteschlange angefügt. In jedem Iterationsschritt wird der Schlange die vorderste Ecke entnommen und als v_h verwendet. Die Iteration endet, wenn die Warteschlange leer ist.

Beispiel 2.6 *Angewandt auf den Graphen aus Abschnitt 2.2 erhalten wir für die von $\diamond$ ausgehenden kürzesten Wege dieselbe Kantenmenge wie beim Dijkstra-Algorithmus. Falls bei der Iteration die Nachbarn v_j von v_h stets in aufsteigender Numerierung behandelt werden, kommen die Ecken in der Reihenfolge*

$$v_1,\ v_2,\ v_6,\ v_3,\ v_5,\ v_4$$

in die Warteschlange.

Der FIFO-Algorithmus kann auch gerichtete Graphen mit negativen Kantenlängen bearbeiten, solange es dort keinen Kreis negativer Gesamtlänge gibt. Beim FIFO-Algorithmus werden mehr Kanten inspiziert als beim Dijkstra-Algorithmus, dafür entfällt die wiederholte Minimumsuche im Vektor *Entf*. Er

ist schneller als der Dijkstra-Algorithmus, falls in dem vorgelegten Graphen nur wenige der $|V| \cdot |V-1|$ möglichen gerichteten Kanten existieren. Als Präzisierung der Angabe „wenig“ findet sich in der Literatur die Schranke

$$|E| \leq 0.3 \cdot |V| \cdot |V-1|.$$

2.4.3 Der Floyd-Warshall-Algorithmus

Für gerichtete Graphen mit beliebigen Kantenbewertungen liefert der Algorithmus von Floyd und Warshall kürzeste Wege zwischen allen Eckenpaaren, falls der Graph keine Kreise mit negativer Kantenbewertungssumme besitzt. Andernfalls findet er einen solchen Kreis. Die Komplexität dieses Algorithmus ist $O(|V|^3)$.

Der Algorithmus arbeitet mit zwei $|V| \times |V|$-Matrizen, der Distanzmatrix $D = (d_{ij})$ und der Transitmatrix $T = (t_{ij})$. Am Ende des Algorithmus gibt d_{ij} die Länge des kürzesten Weges von v_i nach v_j an, und t_{ij} benennt die Ecke mit der höchsten Nummer auf diesem Weg (hierdurch läßt sich der Weg rekonstruieren).

Algorithmus 2.5 *Setze*

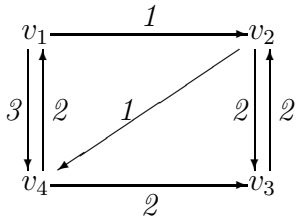
$$\begin{aligned} d_{ij} &:= \begin{cases} l_{ij} & \text{für } v_iv_j \in E \text{ und } i \neq j \\ 0 & \text{für } i = j \\ \infty & \text{sonst} \end{cases} \\ t_{ij} &:= 0 \end{aligned}$$

Für $j = 1, \ldots, |V|$*:*

- *Für* $i = 1, \ldots, |V|;\ i \neq j$*:*
  - *Für* $k = 1, \ldots, |V|;\ k \neq j$*:*
    - *Setze* $d_{ik} := \min\{d_{ik}, d_{ij} + d_{jk}\}$.
    - *Falls* d_{ik} *verändert wurde, setze* $t_{ik} := j$.
  - *Falls* $d_{ii} < 0$ *ist, brich den Algorithmus vorzeitig ab. (Es wurde ein Kreis negativer Länge gefunden.)*

Dieser Algorithmus läßt sich zwar ohne größere Schwierigkeiten effizient programmieren, das Rechnen mit Papier und Bleistift wird aber schon bei niedrigen Eckenanzahlen recht ermüdend. Deshalb sind die folgenden Beispiele besonders klein gewählt:

Beispiel 2.7 *1. Für diesen Graphen mit vier Ecken*



lauten die Matrizen D und T zu Beginn des Algorithmus:

$$D = \begin{pmatrix} 0 & 1 & \infty & 3 \\ \infty & 0 & 2 & 1 \\ \infty & 2 & 0 & \infty \\ 2 & \infty & 2 & 0 \end{pmatrix}, \quad T = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Für $j = 1$ erhalten wir nach Durchlauf der Schleifen für i und k nachfolgende Matrix[3]*, wobei sich nur im Falle $i = 4, k = 2$ bzw. an der Stelle $d_{4,2}$ etwas geändert hat:*

$$D = \begin{pmatrix} 0 & 1 & \infty & 3 \\ \infty & 0 & 2 & 1 \\ \infty & 2 & 0 & \infty \\ 2 & 3 & 2 & 0 \end{pmatrix}, \quad T = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

In der Matrix T wurde $t_{4,2}$ geändert, da hier der Knoten eingetragen wird (wird durch j identifiziert), der die letzte Änderung bewirkt hat.

Für $j = 2$ erhalten wir nach Durchlauf der Schleifen für i und k nachfolgende Matrix:

$$D = \begin{pmatrix} 0 & 1 & 3 & 2 \\ \infty & 0 & 2 & 1 \\ \infty & 2 & 0 & 3 \\ 2 & 3 & 2 & 0 \end{pmatrix}, \quad T = \begin{pmatrix} 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Am Ende des Algorithmus haben wir die Einträge

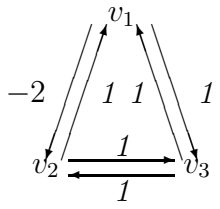
$$D = \begin{pmatrix} 0 & 1 & 3 & 2 \\ 3 & 0 & 2 & 1 \\ 5 & 2 & 0 & 3 \\ 2 & 3 & 2 & 0 \end{pmatrix}, \quad T = \begin{pmatrix} 0 & 0 & 2 & 2 \\ 4 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

[3]Paarungen (i, k) : $(2,2)(2,3)(2,4)(3,2)(3,3)(3,4)(4,2)(4,3)(4,4)$

Dabei bedeutet z.B. der Wert $t_{32} = 0$, daß der kürzeste Weg von v_3 nach v_2 die Kante v_3v_2 ist, während $t_{31} = 4$ bedeutet, daß der kürzeste Weg von v_3 nach v_1 aus dem kürzesten Weg von v_3 nach v_4 gefolgt von dem kürzesten Weg von v_4 nach v_1 besteht. Wegen $t_{34} = 2$ und $t_{32} = t_{24} = t_{41} = 0$ lautet der kürzeste Weg von v_3 nach v_1:

$$v_3,\ v_2,\ v_4,\ v_1$$

2. *Bei dem Graphen mit drei Ecken*



laute die Startwerte für D und T:

$$D = \begin{pmatrix} 0 & -2 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}, \quad T = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Nach dem Durchlauf für den Wert $j = 1$ bricht der Algorithmus ab, denn eine negative Zahl auf der Hauptdiagonalen von D zeigt an, daß ein Kreis negativer Länge gefunden wurde:

$$D = \begin{pmatrix} 0 & -2 & 1 \\ 1 & -1 & 1 \\ 1 & -1 & 0 \end{pmatrix}, \quad T = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

Der Verlauf dieses Kreises wird der Matrix T entnommen.

Bei der Anwendung dieses Algorithmus auf ungerichtete Graphen ist zu beachten, daß beim Ersetzen einer negativ bewerteten ungerichteten Kante durch ein Paar entgegengesetzt gerichteter Kanten zwangsläufig ein Kreis mit negativer Länge entsteht.

Das Festlegen der Werte der Variablen d_{ik} und t_{ik} kann man so interpretieren, daß die Lösung des Problems „kürzester Weg von v_i nach v_k“ aus den Lösungen zweier Teilprobleme, nämlich „kürzester Weg von v_i nach v_j“ und

„kürzester Weg von v_j nach v_k“ zusammengesetzt wird. Ein solcher Gedankengang ist typisch für das Gebiet der „dynamischen Optimierung“ (Näheres hierzu im Buch über „Algorithmen und Datenstrukturen“).

Abschließend sei noch ein Algorithmus für die Transitmatrix angegeben. Da die hier gewählten kleinen Beispiele den Eindruck vermitteln, dort stünde der Vorgänger drin, und in anderen Quellen eine Variante des Floyd-Warshall-Algorithmus vorgestellt wird, die so organisiert ist, dass dort tatsächlich der Vorgänger drin steht: es sei `weg` der Weg zwischen zwei Ecken, der mittels concat aneinander gefügt werden kann. remove entfernt aus einem Weg eine Ecke. Dann:

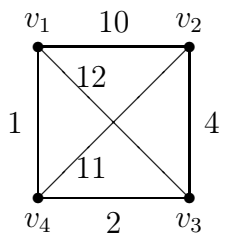
Algorithmus 2.6 *Eingabe ist die Transitmatrix* `T` *und zwei Eckenindizes* `von` *und* `nach` *mit einem Weg als Rückgabewert:* `ermittleWeg(T,von,nach)`

- `zwischen := T[von,nach]`
- *Für* `zwischen == 0`*:* `wegV =<von>; wegN = <nach>`
- *Für* `zwischen > 0`*:*
  - `wegV := ermittleWeg(T,von,zwischen)`
  - `wegN := ermittleWeg(T,zwischen,nach)`
  - `wegV := remove(wegV,zwischen)`
- `return concat(wegV,wegN)`

Beachten Sie: bei nicht zusammenhängenden Graphen oder bei gerichteten Graphen muss im Falle `T[von,nach] == 0` geprüft werden, ob $D[von, nach] \neq \infty$ ist, d.h. ob es tatsächlich eine direkte Verbindung ist.

Aufgaben

1. Führen Sie den Algorithmus von Floyd-Warshall mit nachfolgendem Graphen durch (bis zum Abbruch!). Fertigen Sie eine Dokumentation an, aus der der Ablauf deutlich wird, insbesondere der Verlauf der Laufindizes. Geben Sie am Ende alle kürzesten Wege an, die keine direkten Verbindungen sind.



2. **Floyd-Warshall Algorithmus** (? Punkte)
   Führen Sie den Algorithmus von Floyd-Warshall mit nachfolgendem Graphen durch (bis zum Abbruch!). Fertigen Sie eine Dokumentation an, aus der der Ablauf deutlich wird.

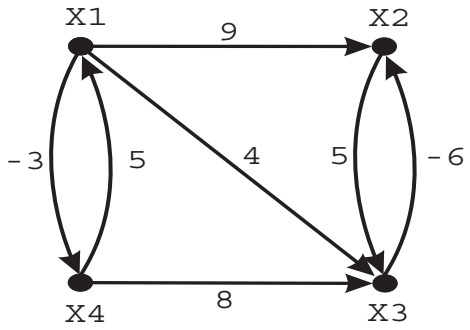


Abbildung 2.3: Kürzeste Wege

3. **Floyd-Warshall Algorithmus** (? Punkte)
   Führen Sie den Algorithmus von Floyd-Warshall mit dem Graphen in Abbildung 2.4 (Seite 60) durch (bis zum Abbruch!). Fertigen Sie eine Dokumentation an, aus der der Ablauf deutlich wird.

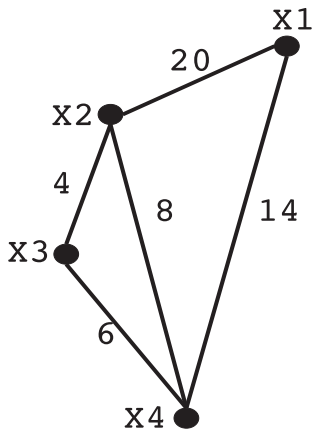


Abbildung 2.4: Kürzeste Wege

4. **Floyd-Warshall Algorithmus** (? Punkte)

(a) Sie haben den Algorithmus von Floyd-Warshall als Alternative zur Bestimmung des kürzesten Weges kennen gelernt. Beschreiben Sie in eigenen Worten, welche Idee dem Algorithmus zugrunde liegt. Worin liegt der wesentliche Unterschied zum Verfahren von Dijkstra ?

(b) Führen Sie den Algorithmus von Floyd-Warshall mit dem gerichteten Graphen in Abbildung 2.5 (Seite 61) durch (bis zum Abbruch!). Fertigen Sie eine Dokumentation an, aus der der Ablauf deutlich wird.

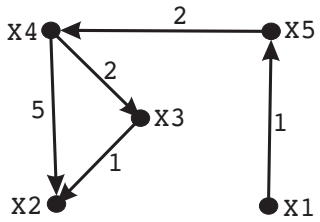


Abbildung 2.5: Kürzeste Wege

(c) Mit Hilfe der Matrix Zwischenknoten ist eine Rekonstruktion des kürzesten Weges zwischen beliebigen Start- und Zielecken möglich. Schreiben Sie diesen Algorithmus in Pseudo-Code.

(d) Die Äußere Schleife des Floyd-Warshall Algorithmus untersucht zuerst alle Zwischenpunkte (Index j). Im Inneren der Schleife werden dann alle Start- (Index i) und alle Zielecken (Index k) abgearbeitet.

In wie fern wirkt sich die Vertauschung der beiden Indizes i und j auf die Korrektheit des Algorithmus aus ? Veranschaulichen Sie dies am Beispielgraphen aus in Abbildung 2.5 (Seite 61).

2.5 Das Problem der längsten Wege

Statt nach der Länge eines kürzesten Weges kann man auch nach der eines längsten Weges zwischen zwei Ecken eines Graphen fragen.

Falls der vorgelegte Graph gerichtet ist und keinen Kreis enthält (dies schließt die Übertragung des Lösungsverfahrens auf ungerichtete Graphen aus), gibt es polynomiale Algorithmen für dieses Problem:

1. Wenn es keine negativen Kantenbewertungen gibt und alle Ecken von der Ausgangsecke v_1 aus erreichbar sind, dann kann eine Variante des

Dijkstra-Algorithmus die längsten Wege von v_1 zu allen anderen Ecken finden. Solche Graphen treten beispielsweise im Zusammenhang mit Netzplänen auf.

2. Unabhängig vom Vorzeichen der Kantenlängen läßt sich das Problem der längsten Wege von einer Ecke in einem gerichteten Graphen zu allen anderen Ecken durch das Verfahren von *Bellman* lösen. Es hat die Komplexität $O(|V|^2)$.

Für das Problem der längsten Wege in einem beliebigen gerichteten Graphen gibt es hingegen (noch) keinen polynomialen Algorithmus.

2.6 Heuristische Methoden: A*-Algorithmus

In diesem Kapitel wurden bisher Tiefensuche und Breitensuche angesprochen sowie mit dem Dijkstra-Algorithmus Suchalgorithmen, die mit exakten Kosten ihre Entscheidung treffen. Ein weiterer alternativer Suchalgorithmus ist der A*-Algorithmus, der den Algorithmus von Dijkstra um eine Abschätzfunktion erweitert und somit als Verallgemeinerung und Erweiterung des Dijkstra-Algorithums betrachtet wird. Falls diese Abschätzfunktion bzw. Heuristik gewisse Eigenschaften erfüllt, kann damit der kürzeste Pfad unter Umständen schneller gefunden werden. Wir stellen also mit dem A*-Algorithmus einen heuristischen Suchalgorithmus vor, der dennoch optimal arbeitet, da er das Konzept von Dijkstra (exakte Kosten bzw. Ecken in der Nähe des Startpunktes bevorzugend) mit dem heuristischen Konzept der Abschätzung (Ecken in der Nähe des Ziels bevorzugend) verbindet. Die Heuristiken versuchen dabei, während der Problemlösung *gute* Ansätze von *schlechten* zu unterscheiden, indem sie die *Kosten* bewerten, etwa durch numerische Zuordnungen. Man unterscheidet hier dabei zwischen der wirklichen, aber unbekannten Funktion und einer Schätzung für diese Funktion. Um eine genaue Analyse zu ermöglichen, werden diese Kosten in einen bereits bekannten Teil (nämlich soweit, wie sie aus der bisherigen Teillösung anfallen) und einen unbekannten Rest aufgespalten. In der folgenden Terminologie wird der Index $*$ stets für die im günstigsten Fall auftretenden Kosten verwandt. Wir wollen hier die Kosten *minimieren*.

Definition 2.3 *Es seien*

$\mathbf{K}$ = *Knotenmenge;*

$\mathbf{T}$ = *Menge der Terminalknoten;*

s = *Startknoten;*

y(n) = *ein Pfad vom Knoten n zu einem Knoten aus T;*

Succ(n) = *Menge der Nachfolger von n;*

k(n, m) = *billigste Kosten von n zu m; dies bleibt undefiniert, falls kein Pfad von n nach m existiert;*

$\mathbf{g_y(n)}$ = *Kosten des Pfades* γ *von s zu n;*

g*(n) = *Kosten des billigsten Pfades von s zum Knoten n;*

g(n) = *Kosten des billigsten bisher bekannten Pfades von s zu n;*

h*(n) = *Kosten des billigsten Pfades von n zu einem Knoten aus T;*

h(n) = *Schätzung der Kosten des billigsten Pfades von n zu einem Knoten aus T;*

f(n) = g(n) + h(n) *Schätzung des billigsten Pfades durch n zu einem Knoten aus T;*

f*(n) = g*(n) + h*(n) *optimale Kosten eines Pfades durch n zu einem Knoten aus T.*

K* = h*(s) = *minimale Kosten überhaupt;*

Wir haben sofort $g(n) \geq g*(n)$ und $f*(s) = g*(s)+h*(s) = 0+h*(s) = K*$.

Dabei ist $g(n)$ keine Schätzung, sondern der bisherige Erfahrungswert, der durch eine bessere Lösung des Teilproblems von s zu n eventuell verbessert werden könnte. Aber $h(n)$ ist eine echte Schätzung, und in der Güte von h liegt der Wert der Problemlösung verborgen, h beinhaltet auch das *Wissen*. Die Differenz $|h*(n) - h(n)|$ ist die Abweichung der Schätzung vom Wert der wirklichen Restkosten, und sie sollte deshalb möglichst klein gehalten werden. Bei der Schätzung kann man im Prinzip zwei Strategien verfolgen:

Definition 2.4 *1.* **Pessimistische** *Strategie des vorsichtigen Hausvaters :* $h*(n) \leq h(n)$ *für jeden Knoten n;*

2. **Optimistische** *Strategie:* $h(n) \leq h*(n)$.

Dazwischen kann es jedoch noch beliebige Mischungen geben. Häufig wird man auch über die Art der Abweichung gar keine genauen Aussagen machen können. Für praktische Anwendungen ist aber sicherlich wichtig, einmal die Differenz $|h*(n) - h(n)|$ möglichst klein zu halten und zum anderen wenigstens tendenzmäßig zu wissen, ob man vorsichtig oder optimistisch schätzt. Für methodische und theoretische Überlegungen eignen sich jedoch die optimistischen Schätzungen weitaus besser. Deshalb werden wir sie in unserer Darstellung auch bevorzugen.

Eine grundlegende Klasse von Algorithmen wird unter dem Namen A*-Algorithmus (oder besser: die A*-Algorithmen) zusammengefaßt, welcher die Schätzfunktion h und die Kostenfunktion g als Parameter hat. Für optimistische Schätzungen hat er spezielle Eigenschaften. Der Algorithmus gibt dabei nicht nur die Auswahl des nächsten Knotens an, sondern führt auch in einer Liste WEG(n) Buch über den Weg, der bis zum jeweils aktuellen Knoten n führte. Mit + bezeichnen wir dabei sowohl die Addition als auch die Listenkonkatenation; das Streichen des Elementes n aus LISTE notieren wir durch LISTE\(n) (Mengenminus).

Algorithmus 2.7 *Der A*-Algorithmus*

1. *Setze g(s) = 0, CLOSED = [], OPEN = [s] , WEG(s) = [s] und berechne h(s) sowie f(s) = g(s) + h(s).*
2. *Wenn OPEN = [], dann STOP mit negativem Ausgang (d.h. es wurde keine Lösung gefunden).*
3. *Wenn OPEN ≠ [], dann wähle n aus OPEN mit f(n) minimal, setze OPEN = OPEN\(n) , CLOSED = [n] + CLOSED.*
4. *Wenn n ∈ T, dann STOP mit positivem Ausgang (mit Gesamtkosten g(n)).*
5. *Wenn n ∉ T, dann betrachte alle n' ∈ Succ(n):*
   (a) *Wenn n' weder auf OPEN noch auf CLOSED, dann setze*
   WEG(n') = [n'] + WEG(n), OPEN = [n'] + OPEN, g(n') = g(n) + k(n,n')
   und berechne f(n') = g(n') + h(n').
   (b) *Wenn n' auf OPEN oder CLOSED und g(n) + k(n,n') < g(n'), dann setze*
   WEG(n') = [n'] + WEG[n] und g(n') = g(n) + k(n,n').
   Falls speziell n' ∈ CLOSED, dann setze
   OPEN = [n'] + OPEN und CLOSED = CLOSED\(n').

6. Gehe zurück zu 2..

Bemerkung 2.1 *An den Knoten sind nur Informationen über die verbleibenden Restprobleme notiert, weshalb ein Knoten auch auf verschiedene Weise (mehr oder weniger umständlich) erreicht werden könnte. Man kann also einen in 5. neu erzeugten Nachfolgerknoten eventuell auch schon auf OPEN oder CLOSED finden. Der Wert g(n) wird dynamisch berechnet und hängt von den bisher untersuchten Wegen zu n ab. Realisiert man die Listen OPEN und CLOSED als Stacks, dann werden die neuen Knoten jeweils oben auf dem Stack abgelegt. Das bedingt aber in 3. bzw. gegebenenfalls auch in 5. ein Durchsuchen des Stacks und die Entfernung eines anderen als des obersten Stackelements. Es kann daher zweckmäßig sein, bei Vorliegen einer bestimmten Funktion* `f` *die Listen OPEN und CLOSED in anderer Reihenfolge aufzubauen; Stack ist für eine Prioritätswarteschlange eine ungeeignete Datenstruktur.*

Im Abbildung 2.6 (Seite 65) sind die Funktionen h und k dargestellt, um einen Ablauf des Algorithmus zu zeigen. Im Abbildung 2.7 (Seite 67) ist der

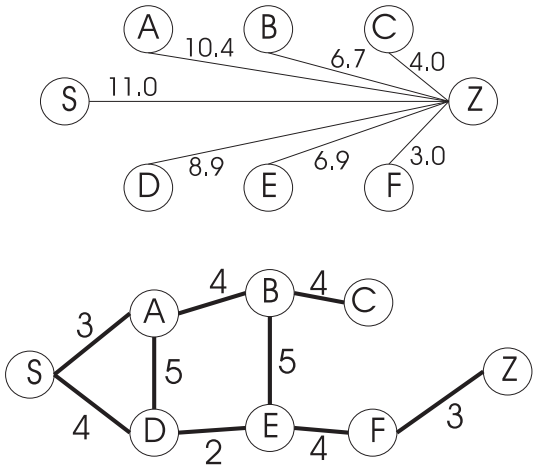


Abbildung 2.6: Schätzfunktion `h` und Kostenfunktion `k`

Ablauf des Algorithmus aufgezeigt. Die Berechnung der Funktion `f` ist unterhalb der Knoten aufgezeigt. In Schritt 2 wird der neue Knoten `A` nicht in die OPEN-Liste aufgenommen, da dort bereits `A` mit geringeren Kosten `g(A)` vorhanden ist.

Es wurde bereits erwähnt, daß der A*-Algorithmus die Funktionen g und h als Parameter enthält. Wir wollen versuchen, anhand von Beispielen ein Gefühl für die hierin verborgene Bandbreite zu erhalten.

2.6.1 Breitensuche in einem Baum

Man wähle k(n,n') = 1 und h(n) = 0 für alle n und n' ∈ Succ(n). Dann sind die Kosten g(n) gerade die Tiefe des Knotens n und die Knoten gleicher Tiefe werden zuerst untersucht.

Eine leichte Verallgemeinerung der Breitensuche besteht darin, daß die Forderung k(n,n') = 1 aufgegeben wird. Dann werden zuerst alle diejenigen Knoten abgearbeitet, die mit den gleichen Kosten von der Wurzel aus erreicht werden können. Man spricht hier auch von der (Brunch-and-Bound) Strategie der *uniformen Kosten*. Dieses Verfahren soll im folgenden Beispiel bezogen auf die Aufgabenstellung in Abbildung 2.6 (Seite 65) dargestellt werden. Der Weg wird dabei nicht notiert.

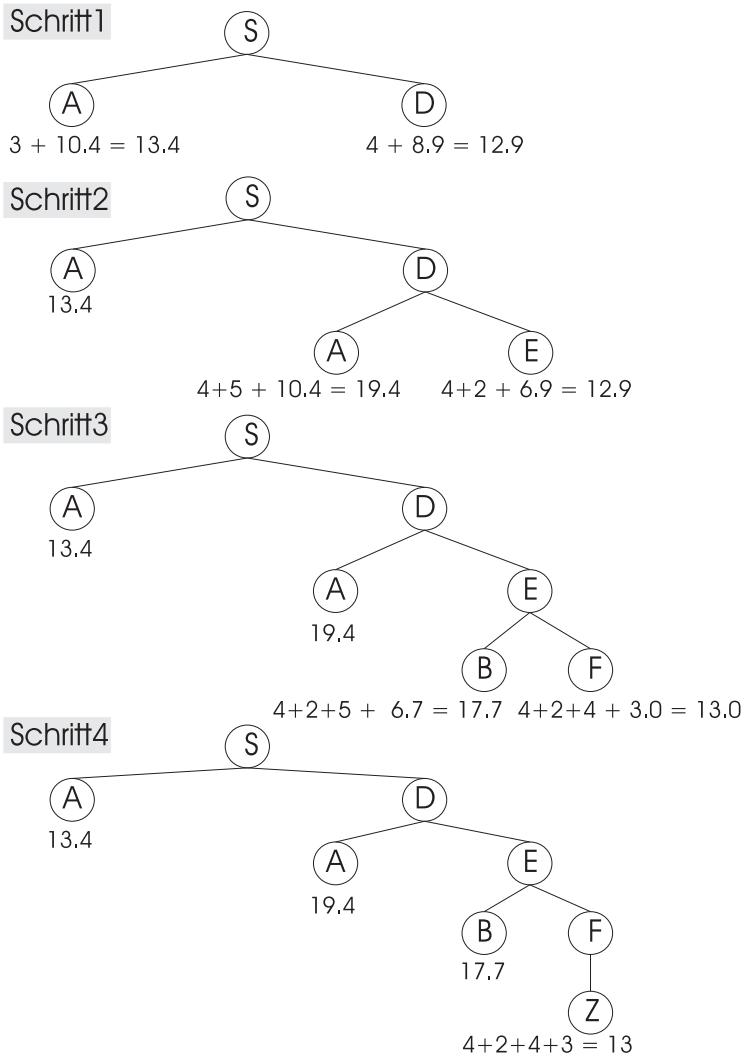


Abbildung 2.7: Ablauf A*-Algorithmus

OPEN	CLOSED	Expandiert
S(0)		A(0 + 3); D(0 + 4);
A(3), D(4)	S(0)	B(3 + 4); D(3 + 5);
D(4), B(7)	S(0),A(3)	A(4 + 5); E(4 + 2);
E(6), B(7)	S(0),A(3),D(4)	B(6 + 5); F(6 + 4);
B(7), F(10)	S(0),A(3),D(4),E(6)	C(7 + 4); E(7 + 5);
F(10), C(11)	S(0),A(3),D(4),E(6),B(7)	Z(10 + 3);
C(11), Z(13)	S(0),A(3),D(4),E(6),B(7),C(11)	

```
Z(13)
```

2.6.2 Tiefensuche in einem Baum

Wir setzen k(n,n') = -1 und h(n) = 0 für alle n und n' ∈ Succ(n). Dann sind die Kosten g(n) gerade die Tiefe des Knotens n mit negativem Vorzeichen und die tiefsten Knoten werden zuerst untersucht.

Backtracking ist eine weitere Modifikation dieser Tiefensuche. Sie wurde essentiell bei der Interpretation von Prolog-Programmen benötigt. Wir beschreiben sie nur informell. Dabei betrachten wir gleich den Fall, daß die Suche nach der Tiefe noch beschränkt sein soll, also eine Schranke S hierfür angegeben ist. Die Funktion f mißt wieder die Tiefe (mit negativen Zahlen). Es wird nun der erste Knoten n aus OPEN gewählt; wenn seine Tiefe S überschreitet oder alle von ihm ausgehenden Zweige untersucht wurden, wird er von OPEN entfernt. Sonst wird ein (neuer) Knoten n' ∈ Succ(n) gewählt, auf WEG seine Relation zu n vermerkt. Wenn n' Zielknoten ist, hat man einen positiven Abbruch; falls dies nicht der Fall ist, wird n' bei Succ(n') = ∅ von OPEN entfernt und die Prozedur rekursiv aufgerufen, bei Succ(n') ≠ ∅ wird ein Nachfolger ausgewählt.

2.6.3 Aussagen über den A*-Algorithmus

Satz 2.1 *Wenn überhaupt eine Lösung existiert (d.h. $T \neq \emptyset$), dann terminiert der A*-Algorithmus mit einem $t \in T$. Für optimistisches h liefert der Algorithmus bereits eine optimale Lösung.*

Beweis Wir nehmen einen Knoten t ∈ T und einen billigsten Pfad γ von s zu einem t ∈ T und betrachten den A*-Algorithmus zu einem beliebigen Zeitpunkt. Die erste Überlegung ist, daß stets ein Knoten n ∈ OPEN auf γ liegt. Das folgt durch Induktion über die Anzahl der Schritte von A*. Daraus folgt bereits, daß A* höchstens an einem n ∈ T terminieren kann, weil sonst OPEN = [] sein müßte. Die zweite Überlegung ist, daß für den ersten solchen Knoten n_0 auf γ (für den also alle früheren bereits auf CLOSED liegen) stets $g(n_0) = g^*(n_0)$ gilt. Dies folgt aus dem Schritt 5. des A*-Algorithmus. Damit gilt dann $f(n_0) = g^*(n_0) + h(n_0) \leq g^*(n_0) + h^*(n_0) = f^*(n_0)$. Weil nun γ ein optimaler Pfad war, gilt, daß seine Kosten gerade K* sind; es folgt daher $f^*(n_0) = K^*$. Jetzt nehmen wir an, A* terminiere mit t ∈ T und $f(t) = g(t) > K^*$. Nach Definition erfüllte t bei seiner Wahl die Bedingung $f(t) \leq f(n)$ für alle n ∈ OPEN. Das

widerspricht aber unserer gerade angestellten Überlegung. Die Terminierung von A* erfolgt deshalb, weil sonst ein Knoten unendlich oft wieder geöffnet werden müßte.

Eine andere Formulierung der dem letzten Beweis zugrundeliegenden Überlegungen ist in folgenden Eigenschaften zusammengefaßt, die man sich im einzelnen klarmachen möge.

1. Wenn ein Knoten n expandiert wird, dann gilt f(n) $\leq$ K*.
2. Jeder Knoten n $\in$ OPEN mit f(n) $<$ K* wird auch tatsächlich expandiert.
3. A* schaltet Knoten n mit f(n) $>$ K* endgültig aus.

Der sicherste Weg, optimistisch zu schätzen, ist, stets h(n) = 0 zu setzen. Das geschieht jedoch auf Kosten eines hohen Arbeitsaufwandes, denn h beinhaltet keinerlei Informationen über das Suchproblem.

Definition 2.5 *Für zwei optimistische Schätzfunktionen h_1 und h_2 heißt h_2* **besser informiert** *als h_1, falls $h_1(n) < h_2(n)$ für jeden Knoten $n \in T$ gilt.*

Man sagt jedoch, das h_2 nicht schlechter informiert ist als h_1, wenn stets $h_1(n) \leq h_2(n)$ gilt. Am schlechtesten informiert (wenn auch am optimistischsten) ist h(n) = 0.

Wir vermerken noch, dass es bei großen Anzahlen auf wenige Knoten i.allg. nicht ankommt; man verallgemeinert die Begriffsbildungen dann derart, dass sie im statistischen Mittelwert gelten.

Diese Definition hat ein erwartetes Resultat zur Folge:

Satz 2.2 *Wenn $A_1{}^* = A_1{}^*(h_1)$ und $A_2{}^* = A_2{}^*(h_2)$ zwei A*-Algorithmen mit den optimistischen Schätzfunktionen h_1 und h_2 sind, wobei h_2 besser informiert als h_1 ist,dann wird jeder Knoten, der von $A_2{}^*$ expandiert wird, auch von $A_1{}^*$ expandiert ($A_2{}^*$ arbeitet also in diesem Sinne nicht schlechter als $A_1{}^*$).*

Beweis Das erste Argument ist, dass ein Knoten n beim A*-Algorithmus sicher dann expandiert wird, wenn es einen Weg γ von s zu n gibt, so dass für jeden Knoten m auf γ die Ungleichung f(m) $<$ K* gilt. Andernfalls betrachtet man den ersten Knoten m von OPEN, für den das nicht der Fall ist und erhält einen Widerspruch. Der zweite Punkt ist, dass es für die Expansion von n auch einen Pfad γ geben muß, auf dem für jeden Knoten m wenigstens f(m) $\leq$ K* gilt. Dies erledigt man durch Induktion über die Knoten auf der Liste WEG(n). Damit haben wir aber auch sofort die Behauptung bewiesen.

Eine Kritik bei der Betrachtungsweise ist, dass theoretisch auch die Breitensuche auf jeden Fall die Lösung findet, man dies aber u.U. nicht mehr erlebt. Wichtig ist, eine optimistische Funktion h zu finden, die möglichst dicht an h* liegt. Damit wird der Suchraum erheblich eingeschränkt. Optimal wäre eine Schätzfunktion h = h*, weil dann immer der beste Weg ausgewählt würde, a.h. A* ohne Umschweife auf die optimale Lösung zusteuern würde.

Eine weitere Kritik an dieser ganzen Betrachtungsweise ist, dass es eigentlich nicht auf die Anzahl der expandierten Knoten, sondern auf die Anzahl der Expansionen selbst ankommt. Das ist nicht dasselbe, weil ein Knoten mehrmals expandiert werden kann. Eine sinnvolle Begriffsbildung ist jetzt möglich:

Definition 2.6 *h heißt* **monoton**, *falls stets h(n) ≤ k(n,n') + h(n') für n' ∈ Succ(n) gilt.*

Die Monotonie setzt sich, wie man durch Induktion zeigt auch auf die Kosten von n nach n' längs beliebiger Pfade fort. Es gilt:

Satz 2.3 *Monotone Schätzfunktionen sind optimistisch.*

Beweis Man nehme einen Knoten m ∈ T und erhält h(n) ≤ k(n,m) + h(m) = k(n,m) = h*(n).

Die Betrachtung monotoner Schätzfunktionen hat, wie die der optimistischen, ein methodisches Motiv: Man kann ihr Verhalten besser kontrollieren. Ihre wesentlichen Eigenschaften lassen sich wie folgt zusammenfassen:

Satz 2.4 *Für A*= A*(h) mit monotonem h gilt:*

1. *g(n) = g*(n) für all n ∈ CLOSED;*
2. *wenn n nach m expandiert wurde, dann gilt f(n) ≤ f(m);*
3. *wenn n expandiert wurde, dann gilt g*(n) + h(n) ≤ K*;*
4. *jeder Knoten mit g*(n) + h(n) < K* wird expandiert.*

Beweis Wir gehen wie beim Beweis von Satz 2.1 (Seite 68) vor und verwenden einige der dort gemachten Erkenntnisse.

Für 1. betrachten wir einen optimalen Pfad γ zu n und nehmen an, es sei g(n) > g*(n). Dann muß OPEN mindestens einen weiteren Knoten von γ enthalten; es sei m der erste solche Knoten. Für m gilt dann g(m) = g*(m) und wir erhalten aus der Monotonie
f(m) = g*(m) + h(m) ≤ g*(m) + k(m,n) + h(n) = g*(n) + h(n) < g(n) +

h(n) = f(n). Das widerspricht aber der Wahl von n als zu expandierendem Knoten.

Für 2. nehmen wir an, n sei direkt nach m expandiert. Wenn sich n und m beide bereits auf OPEN befanden, ist die Behauptung klar. Andernfalls muß n ∈ SUCC(M) sein und die Monotonie liefert
f(n) = g(m) + k(m,n) + h(n) ≥ g(m) + h(m) = f(m).

3. ist eine unmittelbare Folge von 1. und den Überlegungen zu Satz 2.1.

Zu 4. betrachten wir wieder einen optimalen Pfad γ zu n; m sei der Vorgänger von n auf γ. Wir erhalten
g*(n) + h(n) = g*(m) + k(m,n) + h(n) ≥ g*(m) + h(m).
Wenn also g*(n) + h(n) < K* ist, so gilt die entsprechende Ungleichung auch für m und durch Induktion schließlich auf ganz γ.

Wir kommen jetzt zu dem Problem zurück, eine möglichst gut informierte optimale Schätzfunktion h zu finden. Die Vorgehensweise, die wir zu diesem Zweck besprechen wollen, hat im Prinzip einen sehr allgemeinen Charakter; sie heißt die Methode der *relaxierten Modelle*. In jedem Modell M sind unter gewissen, in der Beschreibung von M festgelegten Voraussetzungen Inferenzen oder Aktionen möglich. Lockert man diese Voraussetzungen, so erhält man ein anderes Modell M', welches (relativ zu M gesehen) *relaxiert* heißt. Der vernünftige Gebrauch dieser Vorgehensweise besteht darin, M' berechnungsmäßig einfacher, aber nicht zu trivial zu wählen. Im Graphenmodell bedeutet die Annahme eines relaxierten Modells, dass neue Kanten eingeführt werden, denn es sind ja gelegentlich neue Übergänge möglich. Den erweiterten Graphen wollen wir auch als relaxiert bezeichnen; seine Verwendung im A*-Algorithmus (oder seiner Variation davon) geschieht nun so: Die im relaxierten Graphen (echten) billigsten Kosten h'(n) von n zu einem Knoten aus T werden als Schätzung für h*(n) im ursprünglichen Graphen verwandt.

Im relaxierten Graphen haben wir auch eine neue Kostenfunktion k'(n,m), für die natürlich k'(n,m) ≤ k(n,m) gilt. Außerdem gilt h'(n) ≤ k'(n,m) + h(m), weil es sich im relaxierten Modell M' um die tatsächlich optimalen Kosten handelt, woraus wir schließlich h'(n) ≤ k(n,m) + h(m) erhalten, h' ist also eine monotone Schätzfunktion! Häufig wird man aber auch h' selbst nur schätzen können, dann ist die Monotonie natürlich nicht mehr garantiert.

Bei einer komplexeren Konfigurationsaufgabe könnte man nun auf folgende Weise vorgehen:

1. Man schätze die Kosten für die noch zu konfigurierenden Teile ab. Dazu benötigt man eine Liste für die minimalen erforderlichen Kosten; diese

definiert das relaxierte Modell und liefert eine optimistische Schätzfunktion h.

2. Man informiere h dadurch besser, dass man Constraints aufgrund bisher getroffener Entscheidungen zusammenstellt, welche bestimmte (billige, aber nicht mehr zugelassene) Lösungen ausschalten.

2.6.4 Dijkstra-Algorithmus

Verfügt man über eine Implementierung des A*-Algorithmus kann man mit der Heuristik $h(n) = 0$ den Dijkstra-Algorithmus implementieren. Die Variable OK wird dann durch die OPEN-Liste ($OK = false$) und die CLOSED-Liste ($OK = true$) abgebildet.

Verfügt man dagegen über eine Implementierung des Dijkstra-Algorithmus kann man bei einer monotonen Heuritsik diese in die Kostenfunktion einrechnen und realisiert damit den zugehörigen A*-Algorithmus.

Verfügt man über eine Implementierung des A*-Algorithmus kann man mit der Kostenfunktion $g(n) = 0$ einen Greedy-Algorithmus implementieren. Greedy sucht den durch eine lokale Bewertung vorgenommene, also die Heuristik $h(n)$, günstigsten Nachfolger im Sinne einer gierigen Vorgehensweise.

Kapitel 3

Bäume und Wälder

Eine spezielle Art von Graphen findet besonders häufig Anwendung in der Informatik: die Bäume. Sie werden z.B. verwendet, um syntaktische Strukturen darzustellen (Syntaxbaum), logische Strukturen zu erfassen (Baumtheorie), Hierarchien zu verarbeiten (Subsumptionshierarchie), Abläufe zu verdeutlichen (Tiefen-/Breitensuche) usw. Sie eignen sich insbesondere gut zur Darstellung und Verarbeitung von hierarchisch gegliederten Strukturen, da für jedes Element genau ein Zugriffspfad existiert (Eindeutigkeit des Zugriffpfades).

3.1 Definition und Eigenschaften

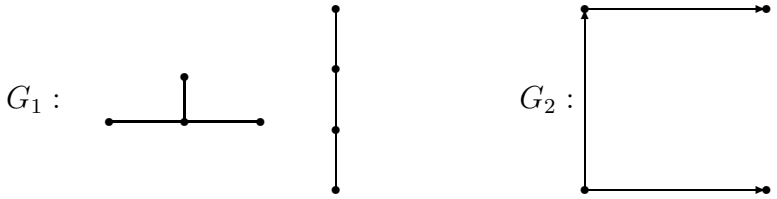
Definition 3.1 (Baum, Wald, azyklischer Digraph)

1. *Ein ungerichteter zusammenhängender kreisloser Graph heißt ein* Baum.
2. *Ein ungerichteter Graph, dessen Komponenten Bäume sind, heißt ein* Wald.
3. *Ein gerichteter Graph heißt ein Baum [Wald], wenn der zugrundeliegende ungerichtete Graph ein Baum [Wald] ist*[1].
4. *Ein gerichteter kreisloser Graph heißt* azyklisch[2].

Beispiel 3.1 *In der folgenden Abbildung ist G_1 ein Wald und G_2 ein azyklischer Digraph:*

[1] Im gerichteten Fall sind z.B. die Bäume interessant, in denen die Richtung eine gewünschte Interpretation wiedergibt, so etwa bei den Wurzelbäumen, wenn die Richtung den Weg von der Wurzel zu allen anderen Ecken aufzeigt.

[2] Mitunter wird auch die Bezeichnung DAG als Abkürzung des englischen „directed acyclic graph“ verwendet.



Die hier gewählt Darstellung betont den Umstand, dass i.allg. keine spezielle Art der visuellen Darstellung mit Bäumen verbunden wird.

Azyklische Digraphen (Bäume) werden beispielsweise zur Darstellung hierarchischer Abhängigkeiten bei Mehrfachvererbung in der objektorientierten Programmierung verwendet.

Die wichtigsten Eigenschaften von Bäumen sind in dem nachfolgenden Satz zusammengefaßt.

Satz 3.1 *Sei $G(V, E)$ ein ungerichteter Graph. Dann sind die folgenden Aussagen äquivalent:*

1. *G ist ein Baum.*
2. *Zwischen je zwei verschiedenen Ecken aus G gibt es genau einen Weg.*
3. *G ist zusammenhängend, und jede Kante aus E ist eine Schnittkante.*
4. *G ist zusammenhängend, und es ist $|E| = |V| - 1$.*
5. *G besitzt keinen Kreis, aber durch Hinzufügen einer beliebigen Kante zu E entsteht ein Graph mit genau einem Kreis.*

Beweis: $1 \Rightarrow 2$: Angenommen, es gäbe zwei Wege von der Ecke u zur Ecke v. Dann sei w die Ecke, bei der erstmalig die beiden Wege über unterschiedliche Kanten weitergehen, und es sei x die Ecke, bei der sich die beiden Wege danach erstmalig wieder treffen. Die beiden Teilstrecken der Wege zwischen w und x bilden einen Kreis, was aber nach i) ausgeschlossen ist.

$2 \Rightarrow 3$: Zunächst ergibt sich direkt, dass G zusammenhängend ist; dies folgt daraus, dass je zwei Ecken miteinander verbunden sind. Da jede Kante einen Weg von einer ihrer Endecken zu der anderen darstellt und es nach Voraussetzung nur einen Weg zwischen diesen beiden Ecken gibt, zerstört die Wegnahme dieser Kante den Zusammenhang zwischen ihren Endecken; sie ist also eine Schnittkante.

$3 \Rightarrow 4$: Wenn alle Ecken eines Graphen mindestens den Grad 2 besitzen, kann man jede Kantenfolge beliebig weit fortsetzen, ohne eine Kante

zweimal unmittelbar hintereinander zu passieren. Sobald eine Ecke in dieser Kantenfolge zum zweitenmal auftritt, wurde ein Kreis gefunden. Eine Kante, die zu einem Kreis gehört, kann aber keine Schnittkante sein. Ein Graph, der die Eigenschaft iii) besitzt, muß daher eine Ecke vom Grad 1 besitzen, oder es handelt sich um den Graphen K_1. Wegnahme dieser Ecke und der mit ihr inzidenten Kante ändert nichts an der Eigenschaft iii). Man kann das Wegnehmen also wiederholen, bis der Graph in den K_1 übergegangen ist. Bis dahin wurden aber genausoviele Ecken wie Kanten entfernt, so dass ursprünglich die Anzahl der Ecken um 1 größer gewesen sein muß als die Anzahl der Kanten.
$4 \Rightarrow 5$: Ein zusammenhängender Graph mit $|V| - 1$ Kanten muß wegen $|E| = \sum_{v \in V} d(v)$ mindestens eine Ecke vom Grad 1 haben. Durch Wegnahme dieser Ecke und der mit ihr inzidenten Kante wird diese Eigenschaft nicht zerstört. Das Wegnehmen läßt sich also so oft wiederholen, bis der Graph zum K_1 geworden ist. Falls G einen Kreis enthielte, müßte die Wegnahme von Ecken vom Grad 1 aber spätestens dann enden, wenn nur noch die Ecken des Kreises übrig sind. Also gibt es in G keinen Kreis.
Für den zweiten Teil von 5 gilt dann: Da G zusammenhängend ist, gibt es zwischen je zwei Ecken mindestens einen Weg. Durch Hinzufügen einer neuen Kante entsteht zwischen ihren inzidenten Ecken ein neuer, alternativer Weg und damit insgesamt ein Kreis. In diesem Sinne entsteht durch Hinzufügen einer beliebigen Kante in einen zusammenhängenden Graphen immer mindestens ein Kreis.Falls die neue Kante auf mindestens zwei Kreisen läge, enthielte die Vereinigung dieser Kreise ohne die neu hinzugefügte Kante ebenfalls einen Kreis. Der ursprüngliche Graph G wäre also nicht kreislos.
$5 \Rightarrow 1$: G muß zusammenhängend sein, denn sonst würde durch das Einfügen einer Kante zwischen zwei Komponenten kein Kreis entstehen. q.e.d.

In bestimmten Anwendungen wird eine Ecke eines ungerichteten Baums besonders hervorgehoben, z.B. wird sie sehr oft als Ausgangs- oder Endpunkt der Zugriffsmechanismen auf einen Baum verwendet oder auch, um eine bestimmte visuelle Darstellung zu erhalten.

Definition 3.2 (Wurzel,Wurzelbaum) *1. Sei $G(V, E)$ ein ungerichteter Baum. Um eine Ecke von G in besonderer Weise hervorzuheben, kann sie als* Wurzel *von G bezeichnet werden. G heißt dann ein* Wurzelbaum. *Die Ecken vom Grad 1 heißen* Blätter *des Wurzelbaums.*

2. *Sei $G = (V, E)$ ein gerichteter Baum. Die Ecke $v \in V$ heißt* Wurzel von *G, wenn alle anderen Ecken in V von v aus erreichbar sind.*

   Gilt für ein $v \in V : d_+(v) = 0 \wedge d_-(v) > 0$, so heißt v Blatt.

3. *Die Länge des längsten Weges von der Wurzel aus wird als* **Höhe** *eines Wurzelbaumes bezeichnet. Teuilweise wird dies in der Literatur auch als* **Tiefe** *bezeichnet.*

Beispiele für Wurzelbäume sind Organigramme (= grafische Darstellungen einer betrieblichen Hierarchie) oder Stammbäume.

In einem Wurzelbaum kann die Wurzel auch Blatt sein. Bei gerichteten Bäumen wird klar unterschieden. Blätter sind quasi Sackgassen, wogegen die Wurzel der Punkt „in alle Richtungen“ ist. Alle anderen Ecken sind in diesem Sinne Durchgangsecken.

Bemerkung 3.1 (Konventionen bei Wurzelbäumen) *Bei der visuellen Darstellung eines Wurzelbaumes wird in der Regel die Wurzel im Bild oben angeordnet. Zwei Ecken stehen im Bild auf gleicher Höhe, wenn dieselbe Anzahl von Kanten zwischen jeder von ihnen und der Wurzel liegt. Seien zwei Ecken u, v adjazent miteinander und v läge im Bild höher als u, d.h. der Weg von v zur Wurzel ist um eins kürzer, als der Weg von u zur Wurzel. Dann wird v häufig als „Vater“ und u als „Sohn“ bezeichnet*[3].

3.2 Anwendungsbeispiele

3.2.1 Speicherung totalgeordneter Datensätze

Beispiel 3.2 *Die Deutsche Telekom hat ca. 9000 Ortsnetzkennzahlen vergeben. Durch die alphabetische Ordnung der Ortsnetznamen ist die Menge der Datenpaare (Kennzahl, Ortsnetzname) total geordnet.*

Die Speicherung der Datenpaare in einer linearen Liste erfordert für das Suchen eines von insgesamt n Ortsnetznamen $O(n)$ Inspektionen von Einträgen in der Liste. Die Speicherung des Datenbestandes in einem speziellen Baum ermöglicht ein schnelleres Suchen:

Definition 3.3 (Binäre Bäume) *Ein Wurzelbaum heißt* binär*, wenn die Wurzel einen Grad ≤ 2 besitzt und alle anderen Ecken mit Ausnahme der Blätter*

[3] Die deutsche Sprache erschwert hier eine geschlechtsneutrale Namensgebung. Im englischen Sprachraum sind die Bezeichnungen „parent“ und „child“ dagegen gebräuchlich.

vom Grad 2 oder 3 sind, also maximal zwei Söhne haben. Er wird dann kurz als **binärer Baum** *bezeichnet. Ein binärer Baum heißt* balanciert*, wenn zwischen jedem Blatt und der Wurzel dieselbe Anzahl von Kanten liegt und mit Ausnahme der Blätter jede Ecke genau zwei Söhne besitzt. Von diesen wird einer als* rechter Sohn *und der andere als* linker Sohn *bezeichnet.*

Satz 3.2 *(maximale und minimale Höhe eines Wurzelbaumes)*

1. *Die maximale Höhe eines Wurzelbaumes mit N Ecken ist $N-1$. Dieser Fall tritt ein, wenn der Baum zu einer Liste entartet.*

2. *Die minimale Höhe eines binären Baumes mit N Ecken ist $\lfloor(\log_2 N)\rfloor$[4]. Dieser Fall tritt ein, wenn alle Ecken bis auf die Blätter den Grad 3 haben.*

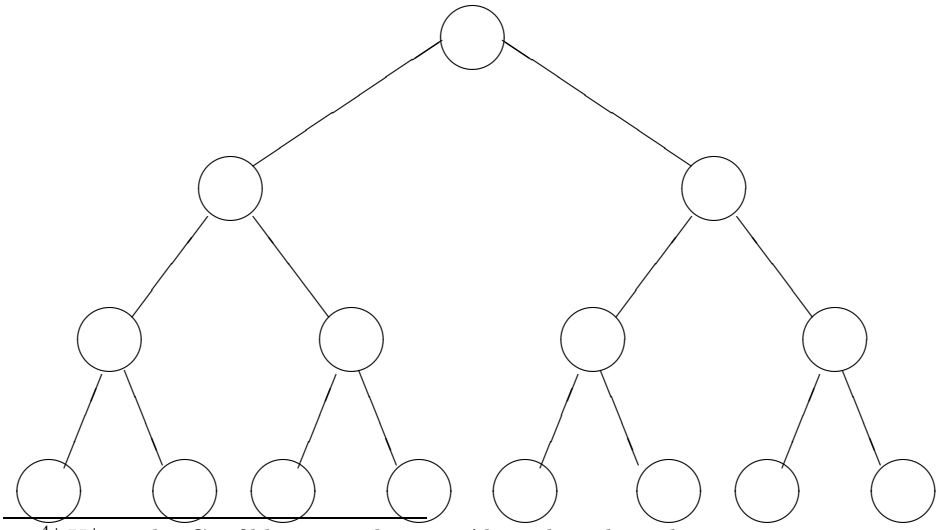
3. *Ein balancierter binärer Baum der Höhe h hat $2^{h+1}-1$ Ecken, wobei h eine natürliche Zahl ≥ 2 ist.*

Ist ein binärer Wurzelbaume mit Höhe h „vollbesetzt“ (und damit balanciert), so hat er

$$1+2+4+8+16+32+\ldots+2^h = 2^{h+1}-1 = N$$

Ecken. In diesem Fall gilt:

$$h = \lfloor \log_2(2^{h+1}-1) \rfloor$$

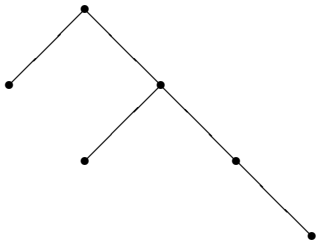


[4] $\lfloor X \rfloor$ ist die Gaußklammer, die eine Abrundung bewirkt.

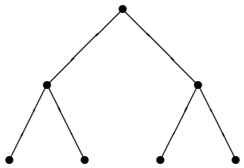
Der vorangegangene binäre Wurzelbaum hat die Höhe 3 und $2^{3+1}-1 = 16-1 = 15$ Ecken; er ist damit „vollbesetzt“ und damit auch balanciert. Umgekehrt gilt für die Höhe: $\lfloor \log_2 15 \rfloor = \lfloor 3.907 \rfloor = 3$

Im Bereich der Algorithmen und Datenstrukturen wird oft eine abgeschwächte Version von *balanciert* verwendet. Im extremen Fall wird nur verlangt, dass alle Blätter zur Wurzel die gleiche Anzahl an Knoten haben.

Beispiele für binäre Bäume sind etwa:



binärer Baum

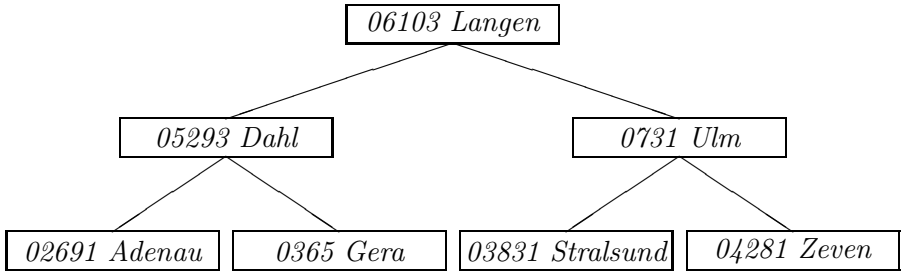


balancierter binärer Baum

Anordnungsprinzip für totalgeordnete Daten auf einem binären Baum

Entsprechend der Ordnungsrelation auf den Datensätzen werden diese den Ecken eines binären Baumes so zugeordnet, dass für jede Ecke v gilt: in dem Teilgraphen, dessen Wurzel der linke Sohn von v ist, stehen nur Datensätze, die kleiner sind als der v zugeordnete Datensatz, und in dem Teilgraphen, dessen Wurzel der rechte Sohn von v ist, stehen nur Datensätze, die größer sind als der v zugeordnete Datensatz.

Beispiel 3.3 *(Fortsetzung) Sieben Ortsnetznamen mit zugehöriger Kennzahl können auf folgende Weise in einem balancierten binären Baum gespeichert werden:*



Satz 3.3 *Die Suche nach einem Datensatz in einem balancierten binären Baum mit n Ecken erfordert die Inspektion von $O(\log(n))$ Datensätzen. (Klar?)*

AVL-Bäume

In diesem Abschnitt soll grob ein Algorithmus vorgestellt werden, der einen Suchbaum als „balancierten“ Suchbaum aufbaut. Wichtig ist hier, dass ein AVL-Baum[5] von Anfang an aufgebaut werden muß, d.h. ein beliebiger Suchbaum kann nur in einen AVL-Baum transformiert werden, indem alle Elemente neu eingefügt werden.

Definition 3.4 *Ein binärer Baum heißt* **AVL-Baum***, wenn für jede Ecke v gilt, dass sich die Höhen des linken und rechten Teilbaums höchstens um 1 unterscheiden.*

Er ist sinnvoll, wenn nach einer aufwendigen Aufbauphase sehr viele Zugriffe auf die eingefügten Daten vorgenommen werden. Die Zugriffszeit ist `O(log(n))`, also die Höhe des binären balancierten Baumes; die Zeit zum balancieren beträgt ebenfalls `O(log(n))`. Löschen ist auch möglich in AVL-Bäumen. Sie werden z.B. von Linux teilweise für die Verwaltung von Speicherplätzen verwendet!

Für diesen Algorithmus werden Rotationen benötigt, die in Abbildung 3.2.1 (Seite 80) dargestellt sind. Kreise stehen dabei für einzelne Ecken, Dreiecke für Teilbäume, der kleine dunkle Kasten für das neu eingefügte Element. Die Balance kann in eine Zahl kodiert werden, die sich wie folgt berechnet: Balance = Höhe rechter Teilbaum - Höhe linker Teilbaum. Zulässige Werte sind also $-1, 0, 1$. Bei anderen Werten muss eine Balancierung vorgenommen werden.

Wichtig sind die Bedingungen, wann eine Rotation vorgenommen werden muß, um den AVL-Baum wieder in Balance zu bringen:

1. **Rechtsrotation**: Die Höhe des Teilbaums `R1` ist um 2 niedriger, als die Höhe des Teilbaums mit Wurzel `B`. Der Unterschied sei durch den Teilbaum `L` begründet.

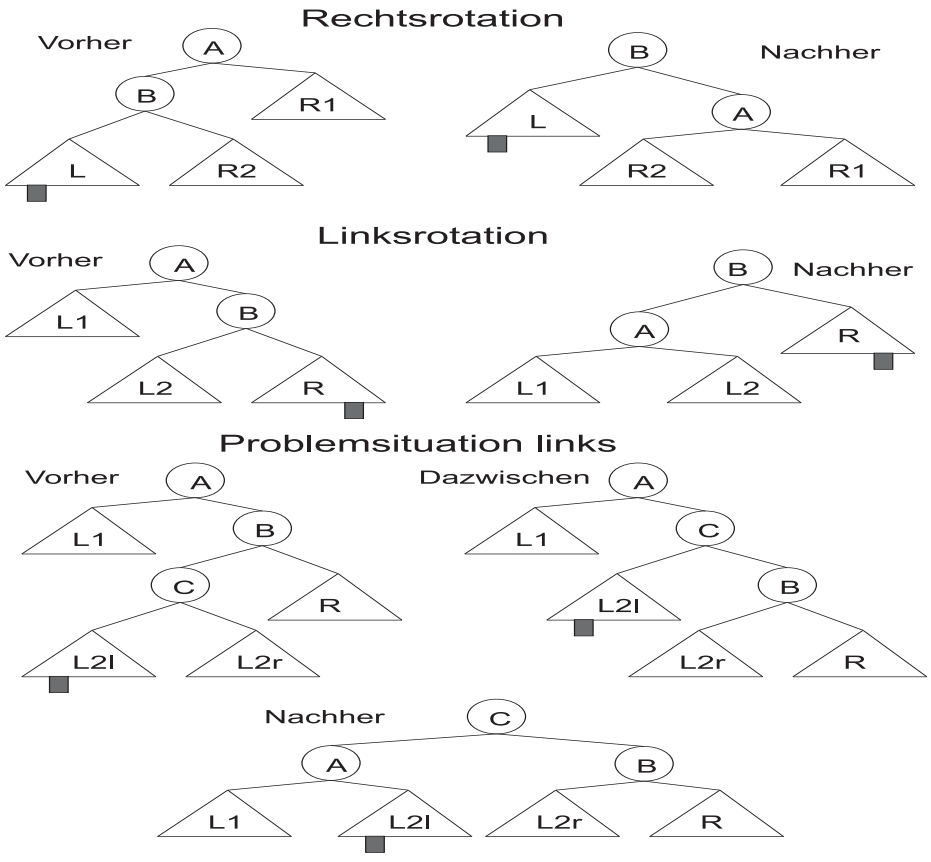
2. **Linksrotation**: Die Höhe des Teilbaums `L1` ist um 2 niedriger, als die Höhe des Teilbaums mit Wurzel `B`. Der Unterschied sei durch den Teilbaum `R` begründet.

3. **Problemsituation links**: (Doppelte Linksrotation) Die Höhe des Teilbaums `L1` ist um 2 niedriger, als die Höhe des Teilbaums mit Wurzel `B`. Der Unterschied sei durch den Teilbaum `L2` begründet. Dieser Teilbaum ist in der Abbildung detaillierter mit Wurzel `C` und den beiden Teilbäumen `L2l` und `L2r` dargestellt. In diesem Fall ist zuerst in dem Teilbaum mit Wurzel `B` eine Rechtsrotation durchzuführen (siehe in der Abbildung

[5] AVL-Bäume sind nach ihren Begründern `A`delson, `V`elskii und `L`andis (1962) benannt.

die Darstellung `Dazwischen`) und dann in dem Baum mit Wurzel `A` eine Linksrotation durchzuführen.

4. **Problemsituation rechts**: (Doppelte Rechtsrotation) Die Höhe des Teilbaums `R1` ist um 2 niedriger, als die Höhe des Teilbaums mit Wurzel `B`. Der Unterschied sei durch den Teilbaum `R2` begründet. Diese Situation ist in der Abbildung nicht dargestellt. Sei dieser Teilbaum detailierter mit Wurzel `C` und den beiden Teilbäumen `R2l` und `R2r` dargestellt. In diesem Fall ist zuerst in dem Teilbaum mit Wurzel `B` eine Linkssrotation durchzuführen und dann in dem Baum mit Wurzel `A` eine Rechtsrotation durchzuführen.



Der Algorithmus selbst arbeitet (im Groben) so, dass er nach dem Einfügen eines Elementes „bottom“ up überprüft, ob eine der vier beschriebenen Situationen vorliegt. Es wird dann eine entsprechende Rotation bzw. Balancierung des Baumes vorgenommen. Diese Bedingung, dass der Algorithmus „bottom

up“ arbeitet ist die zentrale Bedingung für AVL-Bäume! Die Teilbäume in Abbildung 3.2.1 (Seite 80) sind alle AVL-Bäume.

Betrachten wir nun die Höhe des Baumes bzw. seiner Teilbäume für die Abbildung 3.2.1 (Seite 80).

Rechtsrotation *Vorher*: Die Höhe von `R1` sei `n`. Dann ist die Höhe von `R2` gleich `n` oder `n - 1` und die Höhe von `L` gleich `n + 1`. Die Gesamthöhe ist `n + 3`.
Nacher: Die Höhe von dem Teilbaum mit Wurzel `A` ist `n + 1`, also genauso wie die Höhe von `L`! Die Gesamthöhe ist `n + 2`.

Linksrotation *Vorher*: Die Höhe von `L1` sei `n`. Dann ist die Höhe von `L2` gleich `n` oder `n - 1` und die Höhe von `R` gleich `n + 1`. Die Gesamthöhe ist `n + 3`.
Nacher: Die Höhe von dem Teilbaum mit Wurzel `A` ist `n + 1`, also genauso wie die Höhe von `R`! Die Gesamthöhe ist `n + 2`.

Problemsituation links *Vorher*: Die Höhe von `L1` sei `n`. Dann ist die Höhe von `R` gleich `n` oder `n - 1` und die Höhe von `L2` (Teilbaum mit Wurzel `C`) gleich `n + 1`. In der dargestellten Situation ist die Höhe von `L2l` `n`; die Höhe von `L2r` ist dann `n - 1`[6]. Die Gesamthöhe ist `n + 3`.
Nacher: Die Höhe von dem Teilbaum mit Wurzel `A` ist `n + 1` und die Höhe von dem Teilbaum mit Wurzel `B` ist `n` oder `n + 1`. Die Gesamthöhe ist `n + 2`.

Problemsituation rechts Die Situation ist analog, bzw. genaugenommen eine Spiegelung, zu der Problemsituation links.

Balancierter oder nichtbalancierter Baum? – Anmerkungen:

1. Im allgemeinen ist es sehr aufwendig, Datensätze in einem *balancierten* binären Baum anzuordnen. Falls Änderungen am Datenbestand ähnlich häufig vorkommen wie das Suchen nach bestimmten Datensätzen, verzichtet man auf die Balanciertheit des binären Baumes und versucht nur, zu erreichen, dass der entstehende Baum einem balancierten Baum „ähnlicher“ sieht als einer linearen Liste. Im Falle der Liste der Telefonvorwahlen wird jedoch der Datenbestand nach der erstmaligen Festlegung

[6] Wäre die Höhe `n - 2` würde beim Aufbau des AVL-Baumes bereits an dieser Stelle eine Rechtsrotation durchgeführt werden. Die beschriebene Problemsituation würde nicht mehr auftreten!

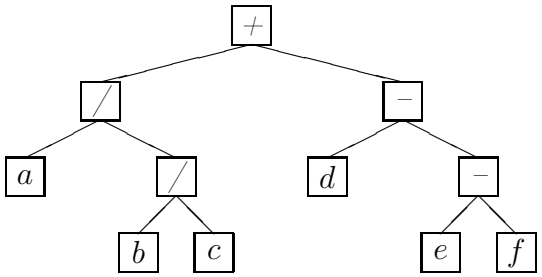
nur noch selten geändert, aber häufig abgefragt. Hier lohnt sich der Aufwand für die Konstruktion eines balancierten Baumes wegen des dadurch erreichten optimalen Suchaufwandes.

2. Ein bewußter Übergang zu einem nichtbalancierten binären Baum bietet sich außerdem immer dann an, wenn die Datensätze mit sehr unterschiedlicher Häufigkeit erfragt werden. Zu häufig aufgerufenen Datensätzen sollten von der Wurzel aus Wege aus wenigen Kanten führen, während selten benötigte Datensätze weiter von der Wurzel entfernt stehen sollten. Das obige Anordnungsprinzip läßt dann in der Regel keinen *balancierten* binären Baum zu.

3.2.2 Auswertung arithmetischer Ausdrücke

Ging es in dem vorangegangenen Beispiel um das Suchen *eines* Datensatzes in einem binären Wurzelbaum, so soll das folgende Beispiel die verschiedenen Möglichkeiten für das Auslesen *aller* in einem solchen Baum gespeicherten Datensätze (Traversierung eines binären Baums) demonstrieren.

Beispiel 3.4 *In einem binären Wurzelbaum seien arithmetische Ausdrücke so gespeichert, dass die Operanden durch die Blätter und die Operatoren durch die übrigen Ecken dargestellt werden. Wir setzen dabei der Einfachheit halber voraus, dass hier nur zweistellige Operatoren (z.B. +, –, *, /) auftreten.*



Zur Auswertung des durch den Baum dargestellten Ausdrucks müssen die Inhalte der Ecken ausgelesen werden. Für das Auslesen sämtlicher Eckeninhalte gibt es drei Möglichkeiten, die sich jeweils durch eine rekursive Prozedur beschreiben lassen:

Im folgenden bezeichne w die Wurzel des vorgelegten Baums, v die jeweils aktuelle Ecke und l_v, r_v den linken, bzw. den rechten Sohn von v.

1. *Inorder:* (symmetrische Reihenfolge) Lies zunächst den Inhalt des Teilbaums mit Wurzel l_v aus, dann den Inhalt von v und dann den Inhalt des Teilbaums mit Wurzel r_v.

2. *Preorder:* (Tiefensuche, Hauptreihenfolge) Lies zunächst den Inhalt von v aus, dann den Inhalt des Teilbaums mit Wurzel l_v und dann den Inhalt des Teilbaums mit Wurzel r_v.

3. *Postorder:* (Nebenreihenfolge) Lies zunächst den Inhalt des Teilbaums mit Wurzel l_v aus, dann den Inhalt des Teilbaums mit Wurzel r_v und dann den Inhalt von v .

Der erste Aufruf jeder dieser Prozeduren erfolgt mit $v := w$.

Für die Breitensuche kann man nicht die Baumstruktur verwenden. Dazu müssen die Ecken z.B. in einer Schlange (Queue) gemäß ihrer Entfernung zur Wurzel angeordnet werden.

In unserem Beispiel erhalten wir den durch den Baum dargestellten arithmetischen Ausdruck je nach dem gewählten Verfahren in

1. Infixnotation: $a/(b/c) + d - (e - f)$

2. Präfixnotation: $+/a/bc - d - ef$

3. Postfixnotation: $abc//def - -+$

Aufgaben

1. Stellen Sie eine betriebliche Hierarchie aus Vorstand, Hauptabteilungsleitern, Abteilungsleitern, Gruppenleitern und Sachbearbeitern als Wurzelbaum dar.

2. Beweisen Sie: Ein (ungerichteter) Baum (mit Ausnahme des K_1) hat mindestens 2 Ecken vom Grad 1.

3. Man kann zeigen, dass ein ungerichteter Graph genau dann bipartit ist, wenn jeder seiner Kreise aus einer geraden Anzahl von Kanten besteht. Benutzen Sie diese Aussage, um zu beweisen, dass jeder Baum ein bipartiter Graph ist.

4. **Bipartit** (? Punkte)
   Zeigen Sie, dass jeder Baum mit mindestens zwei Ecken ein bipartiter Graph ist. Tipp: Entwickeln Sie einen Algorithmus, der aus einem Baum einen bipartiten Graphen macht. Zeigen Sie dann, dass für einen beliebigen Knoten niemals die Konfliktsituation entsteht, dass er beiden Mengen des bipartiten Graphen zugeordent werden könnte.

5. Ordnen Sie die Nachnamen von 7 Mitgliedern Ihrer Semestergruppe in einem balancierten binären Baum an. Fügen Sie dann für jedes weitere Mitglied (in zufälliger Reihenfolge) eine weitere Ecke als Sohn eines der jeweiligen Blätter an. Beachten Sie dabei das Anordnungsprinzip. Beobachten Sie, ob der Baum „nahezu balanciert“ bleibt oder in Richtung auf eine lineare Liste „entartet“. (Stichwort: AVL-Bäume)

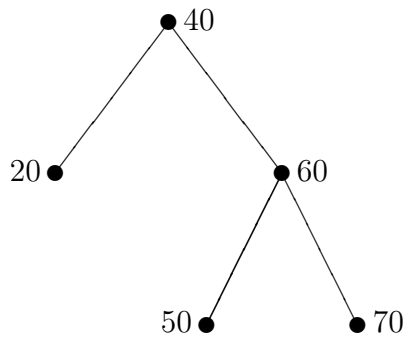
6. Auf welche Weise müssen Sie einen binären Baum, in dessen Ecken Daten unter Beachtung des Anordnungsprinzips gespeichert sind, durchlaufen, um die Daten in der Reihenfolge der auf ihnen definierten Ordnungsrelation auszulesen?

7. Zeichnen Sie alle Bäume mit jeweils acht Ecken, von denen genau eine Ecke den Grad fünf hat. Begründen Sie, warum es nicht noch mehr Bäume gibt. Isomorphe Bäume gelten als eine Lösung!

8. **Sechs Ecken** (? Punkte)
   Zeichnen Sie alle Bäume mit sechs Ecken. Begründen Sie, warum es nicht noch mehr Bäume gibt. Isomorphe Bäume gelten als eine Lösung!

9. Fügen Sie folgende Zahlen **in der vorgegebenen Reihenfolge** in einen leeren AVL-Baum ein: $(4, 5, 8, 6, 9, 7, 1, 0, 2, 3)$. Geben Sie die Anzahl der durchgeführten Rechts- und Linksrotationen an. Geben Sie am Ende den Baum in Inorder als eine Zeile aus.

10. **AVL-Baum** (? Punkte)
    Gegeben sei nachfolgender AVL-Baum:



    Fügen Sie folgende Zahlen **in der vorgegebenen Reihenfolge** in diesen AVL-Baum ein: $(42, 43, 44, 45, 46, 47)$. Geben Sie die Anzahl der durchge-

führten Rechts- und Linksrotationen an. Geben Sie am Ende den Baum in Preorder in einer Zeile aus.

11. **AVL-Baum** (? Punkte)
In dieser Aufgabe ist ein AVL-Baum zu erstellen. Fügen Sie in der folgenden, vorgegebenen Reihenfolge die Elemente in einen leeren AVL-Baum ein. Als Ordnungsrelation gilt die lexikographische Ordnung. Erläutern Sie, wann welche Rotationen stattfinden (Der Ablauf des Einfügens und Rotierens muss klar sein)! Zeichnen Sie den Baum nach jeder (Doppel-)Rotation neu auf.

Söndre_Strömfjord, Berlin, Casablanca, Dakar, El_Salvador, Oslo, Windhuk

12. **Vollständige Induktion** (? Punkte)
Wir betrachten einen balancierten binären Baum der Höhe k. Der **Abstand** zwischen zwei seiner Blätter l_1 und l_2 wird als die Anzahl der Kanten auf dem Weg von l_1 nach l_2 definiert.

Beweisen Sie mittels vollständiger Induktion (Induktionsanfang, -behauptung und -schritt), dass die Summe aller Abstände $(k-1) * 2^{2k} + 2^k$ für $k \geq 0$ ist.
Beachten Sie: Der Abstand zwischen zwei Blättern l_1 und l_2 wird nur einmal gezählt; Ein binärer balancierter Baum der Höhe k hat 2^k Blätter; und bei einem Beweis per vollständiger Induktion hat sich bei Bäumen bewährt, den Induktionsschritt durch das Zusammenhängen zweier Teilbäume zu realisieren.

13. **strukturelle Induktion** (? Punkte)
Es sei $T_0 = [T_1, T_2]$ ein Binärbaum. Ein Binärbaum T heißt *Teilbaum* des Binärbaums T_0 genau dann, wenn $T = T_0$ oder T Teilbaum eines linken (T_1) oder rechten (T_2) Teilbaums von T_0 ist. T heißt *echter Teilbaum* des Binärbaums T_0 genau dann, wenn T Teilbaum von T_0 ist und $T \neq T_0$ gilt. Die *Höhe* $h(T)$ eines Binärbaums T ist folgendermaßen definiert:

- Für den leeren Binärbaum T ist $h(T) = 0$.
- Für den Binärbaum $T = [T_1, T_2]$ ist $h(T) = 1 + max(h(T_1), h(T_2))$.

Zeigen Sie per struktureller Induktion (ähnlich vollständiger Induktion, jedoch nicht über n, sondern über den strukturellen Aufbau): Die Anzahl der Teilbäume eines Binärbaums T ist $\sharp_{neath}(T) \leq 2^{h(T)+1} - 1$.

Erklären Sie kurz, warum sich hier nur eine Abschätzung nach oben angeben läßt.

3.3 Gerüste

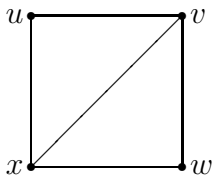
In der Mathematik wie auch in der Informatik kann durch „Verdichtung“ der Informationen die Bearbeitung dieser Informationen erleichtert werden. Der/die LeserIn mögen z.B. an Vektorräume denken, wo mittels einer Basis der ganze Raum beschrieben werden kann. Oder an Flächen, die durch drei Punkte beschrieben werden können. Bei einer Verschiebung einer Fläche müssen nun nicht alle ihre Punkte einzeln verschoben werden, sondern nur die drei sie beschreibenden Punkte.

In ähnlicher Weise kann der Nutzen eines Gerüstes angesehen werden.

Definition 3.5 *Sei $G(V, E)$ ein ungerichteter Graph. Ein Baum $H(W, F)$ heißt ein* Gerüst *von G, wenn H ein Teilgraph von G ist und alle Ecken von G enthält (wenn also gilt $F \subseteq E$ und $W = V$). Wenn H ein Gerüst von G ist, sagt man auch: „G wird von H aufgespannt“.*

Satz 3.4 *Ein Graph G besitzt genau dann ein Gerüst, wenn er zusammenhängend ist.*

Beispiel 3.5 *Der Graph*



besitzt 8 Gerüste (welche?; wieviele davon sind nichtisomorph?).

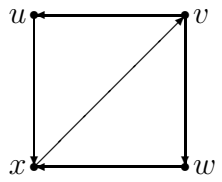
Die Anzahl der verschiedenen (aber nicht notwendig nichtisomorphen) Gerüste des schlingenfreien zusammenhängenden ungerichteten Graphen $G(V, E)$ kann folgendermaßen berechnet werden:

Satz 3.5 *Der Digraph $G'(V, E')$ entstehe aus G dadurch, dass den Kanten aus E eine beliebige, aber feste Richtung gegeben wird. Die Matrix $\tilde{M}$ entstehe aus der Inzidenzmatrix $M(G')$ durch Weglassen einer beliebigen Zeile. Dann gilt:*

Die Anzahl $\tau(G)$ der verschiedenen Gerüste von G ist[7]

$$\tau(G) = \det(\tilde{M} \cdot \tilde{M}^T)$$

Beispiel 3.6 *Im Beispiel 3.5 sei etwa der Digraph G':*



Unter Berücksichtigung der zu den Ecken u, v, w gehörenden Zeilen der Inzidenzmatrix $M(G')$ erhalten wir

$$\tilde{M} = \begin{pmatrix} +1 & -1 & 0 & 0 & 0 \\ -1 & 0 & +1 & -1 & 0 \\ 0 & 0 & 0 & +1 & -1 \end{pmatrix}$$

Damit ergibt sich für die Anzahl der Gerüste[8]*:*

$$\tau(G) = \det(\tilde{M} \cdot \tilde{M}^T) = \det \begin{pmatrix} 2 & -1 & 0 \\ -1 & 3 & -1 \\ 0 & -1 & 2 \end{pmatrix} = 8$$

Eine einfachere Formel ergibt sich für die vollständigen Graphen K_n:

Satz 3.6 (Satz von Caley) *Der vollständige Graph K_n mit n Ecken hat $\tau(K_n) = n^{n-2}$ verschiedene (nicht notwendig nichtisomorphe) Gerüste.*

Ein Beispiel aus der Praxis sind die **Hypercube Netzwerke**: Ein d-dimensionaler Hypercube besteht im Prinzip aus zwei d-1 dimensionalen Hypercubes. In einem d-dimensionalen Hypercube Netzwerk ist jede Ecke mit einer Ecke in jeder Dimension des Netzwerks verbunden. Dabei erhält jede Ecke eine d-bit Binäradresse, wobei jedes Bit eine Dimension repräsentiert. Jede Ecke wird mit jenen Ecken verbunden, deren Adresse sich nur in einem Bit von der eigenen Adresse unterscheidet[9]. Der Durchmesser des Hypercupe

[7] Zur Erinnerung: $\tilde{M}^T$ bezeichnet die transponierte Matrix und $det(\ldots)$ die Determinante einer Matrix.

[8] $det(\ldots) = a_{11}a_{22}a_{33} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{13}a_{22}a_{31}$

[9] Entspricht dem gespiegelten Gray-Code-Verfahren.

beträgt $log_2 N$ bei N Ecken, also nur ein logarithmisches Wachstum bei Erweiterung des Netzes. Außerdem existiert ein „*minimal distance deadlock-free*“ Routing-Algorithmus, der sogenannte *e-cube* oder *left-to-right Algorithmus*. Es wird dabei die Adresse der Start- und der Zielecke XOR verknüpft und die 1-Bits des Ergebnis geben an, welche Dimensionen für das Routing verwendet werden sollen. In jeder Zwischenecke wird dann erneut die aktuelle Eckenadresse mit der Zieleckenadresse XOR verknüpft und so weiter fortgefahren. Die verwendeten Pfade bei der Kommunikation entsprechenden dabei spannenden Gerüsten.

3.3.1 Bestimmung von Minimalgerüsten

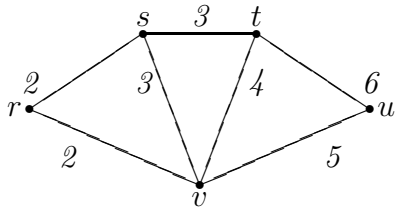
Die wichtigste Aufgabenstellung im Zusammenhang mit Gerüsten verlangt die Bestimmung eines sogenannten Minimalgerüsts.

Aufgabenstellung

Vorgelegt sei ein zusammenhängender schlichter ungerichteter Graph $G(V, E)$, bei dem jede Kante $v_i v_j$ mit einer nichtnegativen Bewertung l_{ij} („Länge“) versehen ist. Gesucht ist ein Gerüst von G mit minimaler Kantenbewertungssumme.

Beispiel 3.7 *Die Gemeinden eines Kreises haben gemeinsam einen Schneepflug angeschafft. Bei Schneefall soll er dafür sorgen, dass zwischen je zwei Gemeinden eine (nicht notwendigerweise direkte) Straßenverbindung erhalten bleibt. Dabei soll die Gesamtlänge der zu räumenden Straßen möglichst klein sein (damit diese Straßen lieber öfters geräumt werden können).*

Das Straßennetz (mit Längenangaben in km) hat folgende Gestalt:



Ein solches Minimalgerüst läßt sich durch den folgenden Algorithmus finden. Die Eingabe besteht aus der Menge E der Kanten mit ihren Längen. Die Ausgabe ist eine Teilmenge F der Kantenmenge (die Eckenmenge eines Gerüsts steht ja von Anfang an fest).

Algorithmus von Kruskal

Algorithmus 3.1 *Eingabe: Eine Menge E der Kanten mit ihren Längen. Ausgabe: Eine Teilmenge F der Kantenmenge.*

- *nummeriere die Kanten $e_1, \ldots, e_{|E|}$ nach steigender Länge. Setze $F := \emptyset$.*
- *Für $i := 1, \ldots, |E|$:*
  - *Falls $F \cup \{e_i\}$ nicht die Kantenmenge eines Kreises in G enthält, setze $F := F \cup \{e_i\}$.*

Beispiel 3.8 *(Fortsetzung) Wir erhalten folgende Sortierung der Kanten:*

$$e_1 = rs,\ e_2 = rv,\ e_3 = st,\ e_4 = sv,\ e_5 = tv,\ e_6 = uv,\ e_7 = tu$$

und das (in diesem Fall eindeutig bestimmte) Minimalgerüst enthält die Kanten

$$F = \{rs,\ rv,\ st,\ uv\}$$

mit der Kantengewichtssumme 12.

Die Korrektheit des Kruskal-Algorithmus ergibt sich aus folgender Überlegung: Sei F die Kantenmenge des durch den Kruskal-Algorithmus gefundenen Gerüsts und $\tilde{F}$ die eines anderen Gerüsts. Sei weiterhin e_j die kürzeste Kante aus $F \setminus \tilde{F}$. $\tilde{F} \cup \{e_j\}$ enthält einen Kreis. Dieser Kreis muß mindestens eine Kante e_k mit $k > j$ enthalten (alle Kanten $e_i \in \tilde{F}$ mit $i < j$ gehören auch zu F, bilden also mit e_j noch keinen Kreis). Durch $\tilde{\tilde{F}} := (\tilde{F} \setminus \{e_k\}) \cup \{e_j\}$ erhalten wir ein Gerüst, dessen Kantenbewertungssumme nicht größer ist als die von $\tilde{F}$. Durch wiederholtes Anwenden dieser Konstruktion führen wir $\tilde{F}$ in F über, ohne dass sich dadurch die Kantenbewertungssumme vergrößert. Also kann die Kantenbewertungssumme von $\tilde{F}$ nicht kleiner gewesen sein als diejenige von F.

Praktische Durchführung des Kantentests

Anhand einer Zeichnung ist leicht festzustellen, ob F durch das Hinzufügen einer bestimmten Kante die Kantenmenge eines Kreises von G enthält. Bei der Überprüfung einer Kante mit Hilfe des Rechners ist die ausdrückliche Suche nach eventuellen Kreisen mit Kanten aus F zu arbeitsaufwendig. Hier hilft die folgende Idee, schneller zum Ziel zu kommen:
– Vor Beginn des Algorithmus werden alle Ecken von 1 bis $|V|$ durchnummeriert.

– Eine Kante darf genau dann zu F hinzugefügt werden, wenn ihre Endecken unterschiedlich nummeriert sind.
– Wenn eine Kante zu F hinzugefügt wird, seien α und β mit $\alpha < \beta$ die Nummern ihrer Endecken. Vor dem Test der nächsten Kante wird die nummerierung aller bisher mit β nummerierten Ecken auf α gesetzt.

Am Ende des Algorithmus tragen dann alle Ecken die Nummer 1. Diese Version kann auch auf nicht zusammenhängende Graphen angewendet werden. Dann bilden jeweils alle nach Beendigung des Algorithmus gleich nummerierte Ecken eine Zusammenhangskomponente.

Komplexität

Bei der Bestimmung der Komplexität des Kruskal-Algorithmus wird üblicherweise das Sortieren der Kanten als eigenes Problem angesehen und deshalb „ausgeblendet“.

Damit besteht der Algorithmus im wesentlichen aus der Inspektion aller Kanten, also aus $O(|E|)$ Schritten. Jeder Schritt umfaßt die Betrachtung der Eckennummern ($O(1)$ Arbeitsgänge), ggf. das Hinzufügen einer Kante zu F ($O(1)$ Arbeitsgänge) und ggf. das Ändern einiger Eckennummerierungen ($O(|V|)$ Arbeitsgänge). Dabei kommt es bei einem zusammenhängenden Graphen genau $(|V|-1)$-mal vor, dass eine Kante zu F hinzugefügt wird. Damit ergibt sich die Zahl der Arbeitsschritte zu:

$$\begin{aligned} \underbrace{(|E|-|V|+1)\cdot O(1)}_{\text{kein Einfügen in } F} + \underbrace{(|V|-1)\cdot(O(V)+O(1))}_{\text{Einfügen in } F} & \\ = \; O(|E|)+O(|V|^2) & \\ = \; O(|V|^2)+O(|V|^2) & \\ = \; O(|V|^2) & \end{aligned}$$

3.3.2 Greedy-Algorithmen

Der Kruskal-Algorithmus ist ein Beispiel für eine allgemeine Klasse von Vorgehensweisen, die als Greedy-Algorithmen bezeichnet werden:

Vorgelegt sei eine endliche Menge E, wobei jedes ihrer Elemente e eine nichtnegative Bewertung $w(e)$ besitzt. Gesucht ist eine Teilmenge $F \subseteq E$ mit minimaler [maximaler] Bewertungssumme $\sum_{e\in F} w(e)$. Dabei werden an F gewisse Bedingungen gestellt, von denen hier nur vorausgesetzt wird, dass, wann immer eine Menge F diese Bedingungen erfüllt, auch jede Teilmenge $F' \subseteq F$ ebenfalls diese Bedingungen erfüllt.

Ein Greedy-Algorithmus geht dann auf folgende Weise vor:

Algorithmus 3.2 *Eingabe: Endliche Menge E.*

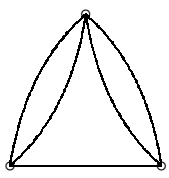
- *Ordne die Elemente von E nach steigender [fallender] Bewertung. Setze $F := \emptyset$.*
- *Für jedes Element $e \in E$ (in der gerade festgelegten Reihenfolge):*
  - *Falls $F \cup \{e\}$ die Bedingungen erfüllt, setze $F := F \cup \{e\}$.*

Die Auswahl des Elementes in Schritt zwei wird lokal optimal vorgenommen, d.h. aus der lokalen aktuellen Sicht ist das nächste kleinste [größte] Element lokal optimal. Ob es auch global optimal ist, also bzgl. der Lösung des gestellten Problems, wird hier nicht geprüft. Die aus lokal optimaler Sicht getroffene Entscheidung wird nicht wieder revediert, deshalb auch der Name Greedy (gierig). Im Unterschied dazu z.B. der Dijkstra Algorithmus: lokal optimal wird in einem Schritt die Distanz zu den benachbarten Ecken bestimmt. Lokal bezieht sich in dem Sinne auf die aktuell untersuchte Ecke und ihre Nachbarn. Da dies aber u.U. nicht global optimal ist, also bezogen auf den ganzen Graphen, wird eine Änderung zugelassen. Die Variable *OK* signalisiert das globale Optimum: wenn diese auf *true* gesetzt wird, ist der global optimal kürzeste Weg, also der tatsächlich kürzeste Weg von der Startecke zu dieser Ecke gefunden. Es ist möglich, die Aufgabenstellungen, für die ein Greedy-Algorithmus die optimale Lösung liefert, mathematisch präzise zu charakterisieren.

Wegen seiner Einfachheit wird ein Greedy-Algorithmus aber auch häufig dazu verwendet, überhaupt erst einmal eine „ganz gute“ Lösung zu finden, von der aus man dann mit Hilfe anderer, aufwendigerer Algorithmen das Optimum ansteuert. Deshalb ist es für die mathematische Forschung beispielsweise ein interessantes Ergebnis, beweisen zu können, dass ein Greedy-Algorithmus für ein bestimmtes Maximierungsproblem immer eine Lösung findet, die mindestens halb so gut ist wie das Optimum. Ein solches Problem ist etwa die ILP-Version des Rucksackproblems, wenn die einzupackenden Gegenstände nach monoton fallenden Werten von c_i/a_i geordnet werden. Eine weitere Aufgabenstellung, auf die das zutrifft, werden wir in Abschnitt 4.3.1 kennenlernen.

Aufgaben

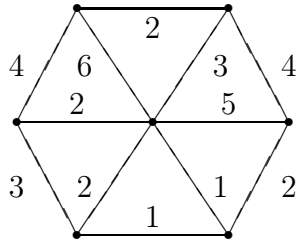
1. Wieviele Gerüste besitzt der folgende Graph?



2. Weisen Sie nach, dass ein Minimalgerüst alle Schnittkanten des vorgelegten Graphen enthalten muß.

3. Führen Sie für das obige „Schneepflug-Beispiel“ den Kruskal-Algorithmus mit Eckennummerierung durch.

4. Bestimmen Sie ein Minimalgerüst des Graphen



5. Zeichnen Sie den K_4. Entfernen Sie Kanten, aber keine Ecken, so dass ein Baum übrig bleibt. Wie viele nicht isomorphe Bäume können Sie auf diese Weise erzeugen ?

6. **Gerüste und mehr** (? Punkte)
Die Fluggesellschaft Graphic-Airlines macht außerhalb der Saison auf allen ihren Strecken Verluste. Sie möchte den Betrieb einschränken, indem sie manche Strecken vorübergehend einstellt. Es soll aber noch jeder Flughafen von jedem anderen aus erreichbar sein, notfalls mit Umsteigen. Aus der Landkarte in Abbildung 3.1 (Seite 92) kann man die Verluste für die einzelnen Strecken entnehmen.

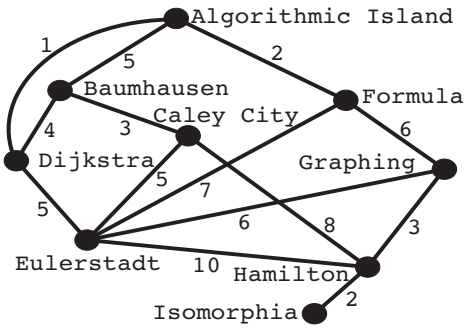


Abbildung 3.1: Verlustreiche Strecken

Sie sollen nun zwei Lösungen erarbeiten. Identifizieren Sie dazu das jeweilige Problem und verwenden einen geeigneten Algorithmus, um das

Problem zu lösen (manuelle Berechnungen werden **nicht** bewertet!). Vergleichen Sie abschließend die beiden Lösungen miteinander und machen Sie der Fluggesellschaft einen Vorschlag (Begründung nicht vergessen!). Hier nun die jeweiligen Anforderungen für eine der beiden gesuchten Lösungen:

(a) Die Gesamtkosten für den Betrieb des Netzes sollen möglichst niedrig sein.

(b) In Eulerstadt ist die Zentrale. Alle Flughäfen sollen von dort aus mit möglichst geringen Verlusten erreichbar sein.

Kapitel 4

Flussprobleme

In diesem Kapitel beschäftigen wir uns grundsätzlich mit schwach zusammenhängenden, schlichten gerichteten Graphen $G(V, E)$ mit $|V| = n$ Ecken. Längs der Kanten wird ein Gut transportiert, etwa Strom, Container etc., und für jede Kante $v_i v_j = e_{ij}$ gibt die *Kapazität* $c(e_{ij}) = c_{ij}$ der Kante an, welche Menge dieses Gutes längs dieser Kante von ihrer Anfangsecke v_i zu ihrer Endecke v_j transportiert werden kann. Aus praktischen Gründen nehmen wir dabei an, dass alle c_{ij} rationale Zahlen sind. Diese Art von Graphen wird in der Literatur auch oft als *Netzwerk* bezeichnet. Beispiele hierfür sind etwa:

- Wenn die Kanten Nonstop-Flugverbindungen zwischen Flughäfen darstellen, dann können die Kapazitäten angeben, wieviele Passagiere pro Zeiteinheit längs dieser Kante befördert werden können.
- Wenn die Kanten Telefonleitungen zwischen Knotenvermittlungsstellen repräsentieren, dann können die Kapazitäten angeben, wieviele Telefongespräche gleichzeitig längs dieser Kante geführt werden können.

Im folgenden werden folgende zwei Mengen verwendet. Für eine Ecke v_i ist $O(v_i) = \{e_{ij} \in E \| s(e_{ij}) = v_i\}$ der *output* dieser Ecke und $I(v_i) = \{e_{ji} \in E \| t(e_{ji}) = v_i\}$ der *input* dieser Ecke. Es gilt $|O(v_i)| = d_+(v_i)$ und $|I(v_i)| = d_-(v_i)$.

Definition 4.1 *Sei $G = (V, E)$ ein schwach zusammenhängender, schlichter gerichteter Graph mit $|V| = n$ Ecken, wobei in G zwei Ecken besonders hervorgehoben seien: eine* Quelle $q := v_1$ *mit $d_-(q) = 0$ und eine* Senke $s := v_n$ *mit $d_+(s) = 0$. Eine Kapazität ist eine Funktion c, die jeder Kante $e_{ij} \in E$ eine positive rationale Zahl (> 0) als Kapaziät zuordnet.*

Ein Fluss *in G von der Quelle $q = v_1$ zu der Senke $s = v_n$ ist eine Funktion f, die jeder Kante $e_{ij} \in E$ eine nicht negative rationale Zahl zuordnet, so dass*

1. *für jede Kante* $e_{ij} : f(e_{ij}) \leq c(e_{ij})$ *gilt (Kapazitätsbeschränkung),*

2. *der gesamte Fluss, der von der Quelle* v_1 *wegtransportiert wird, in vollem Umfang an der Senke* v_n *eintrifft,*

$$\sum_{e_{1j} \in O(q)} f(e_{1j}) = \sum_{e_{in} \in I(s)} f(e_{in})$$

und

3. *für jede übrige Ecke, den sogenannten* inneren Ecken*, werden eintreffende Mengen des Gutes verlustlos weitergeleitet, d.h. es gilt die* Flusserhaltung.

$$\forall j \in \{1, \ldots n\} : \sum_{e_{ij} \in O(v_i), e_{ji} \in I(v_i)} (f(e_{ij}) - f(e_{ji})) = 0$$

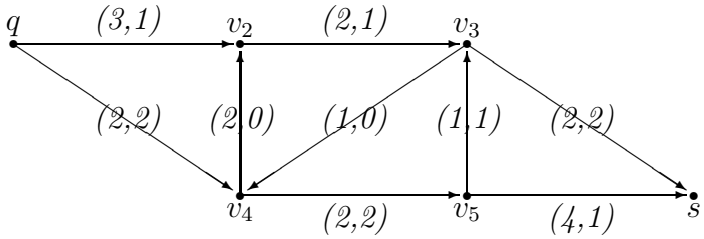
Wir bezeichnen einen solchen Graphen mit einem Fluss als Flussnetzwerk (V, E, f, c).

Der *Wert des Flusses* f ist dann

$$d = \sum_{e_{1j} \in O(q)} f(e_{1j}) = \sum_{e_{in} \in I(s)} f(e_{in})$$

Es ist noch zu bemerken, dass in jedem Graphen es nur genau eine Ecke mit $d_-(q) = 0$ und es nur genau eine Ecke mit $d_+(s) = 0$ gibt, d.h. Quelle und Senke sind eindeutig. Dies liegt an der dritten Forderung für den Fluss f.

Beispiel 4.1 *In dem folgenden Graphen ist jede Kante* $e_{ij} = v_i v_j$ *mit dem Wertepaar* $(c(e_{ij}), f(e_{ij}))$*, d.h. (Kapazität,tatsächlicher Fluss), beschriftet:*



Aus der Quelle fließen 3 Mengeneinheiten ab, in der Senke treffen 3 Mengeneinheiten ein, folglich ist der Wert des Flusses 3. In allen anderen Ecken halten sich eintreffende und abfließende Mengen die Waage, und durch jede Kante fließen höchstens soviele Mengeneinheiten, wie ihre Kapazität es zuläßt.

4.1 Flüsse maximaler Stärke

Bei der ersten Aufgabenstellung dieses Kapitels fragen wir nach der maximalen Menge, die auf diese Weise von der Quelle $q = v_1$ zur Senke $s = v_n$ transportiert werden kann.

4.1.1 Minimale Schnitte

Es sei f der Fluss eines Graphen $G = (V, E)$, und für jede echte Untermenge X der Eckenmenge V von G bezeichne $\overline{X}$ das Komplement von X in V, d.h. $\overline{X} = V \setminus X$.

Wenn X die Quelle q aber nicht die Senke s enthält, sollte intuitiv der Fluss von den Ecken in X zu den Ecken in $\overline{X}$ dem Wert d des gesamten Flusses ($d = f(X, \overline{X}) - f(\overline{X}, X)$) entsprechen.

Beispiel 4.2 *Gegeben sei der Fluss bzw. Graph aus Beispiel 4.1. Es sei (beliebig gewählt)* $X = \{q = v_1, v_2, v_3\}$ *und somit* $\overline{X} = \{v_4, v_5, s = v_6\}$. *Die Menge der Kanten, die* X *und* $\overline{X}$ *verbindet ist* $A(X, \overline{X}) = \{e_{14}, e_{34}, e_{36}, e_{42}, e_{53}\}$. *Es sei nun* $A^+(X, \overline{X}) = \{e_{14}, e_{34}, e_{36}\}$ *die Menge der Kanten aus* X *heraus und* $A^-(X, \overline{X}) = \{e_{42}, e_{53}\}$ *die Menge der Kanten in* X *hinein. Folglich ist der Fluss von den Ecken in* X *zu den Ecken in* $\overline{X}$

$$\sum_{e_{ij} \in A^+(X,\overline{X})} f(e_{ij}) - \sum_{e_{ij} \in A^-(X,\overline{X})} f(e_{ij}) = (2 + 0 + 2) - (0 + 1) = 3$$

Interessant ist auch die Kapazität von X *nach* $\overline{X}$

$$\sum_{e_{ij} \in A^+(X,\overline{X})} c(e_{ij}) = 2 + 1 + 2 = 5$$

Vor der Verallgemeinerung dieser Überlegungen seien folgende Schreibweisen festgelegt:

- Wenn X und Y beliebige Untermengen von Ecken eines Graphen G sind, bezeichnet $A(X, Y)$ die Menge der Kanten, die Ecken aus X mit Ecken aus Y verbinden.
- Wenn g eine beliebige Funktion ist, die den Kanten eines Graphen G nichtnegative rationale Zahlen zuordnet (hier meist die Funktion f oder die Kapazitätsfunktion c), dann ist für zwei beliebige Eckenmengen X, Y von G: $g(X, Y) = \sum_{e \in A^+(X,Y)} g(e)$.

Definition 4.2 *Ein* Schnitt *ist eine Menge von Kanten $A(X, \overline{X})$, wobei $q \in X$ und $s \in \overline{X}$.*

Satz 4.1 *Es sei f ein Fluss in einem Graphen $G = (V, E)$, und es sei d der Wert des Flusses. Wenn $A(X, \overline{X})$ ein Schnitt in G ist, dann gilt $d = f(X, \overline{X}) - f(\overline{X}, X)$ und $d \leq c(X, \overline{X})$.*

Mit anderen Worten ist der gesamte von X herauslaufende Fluss minus dem gesamten in X hineinlaufenden Fluss gleich dem gesamten Fluss d und dieser überschreitet nie die gesamte Kapazität der Kanten von X nach $\overline{X}$.

Der zweite Teil des Satzes besagt, dass der Wert *jedes beliebigen* Flusses kleiner oder gleich der Kapazität der Kanten von X nach $\overline{X}$ für *jeden beliebigen* Schnitt $A(X, \overline{X})$ ist. Ein besonderes Interesse gilt den maximalen Flüssen.

Definition 4.3 *Ein Fluss, dessen Wert*

$$\min\{c(X, \overline{X}) \| A(X, \overline{X}) \textit{ist ein beliebiger Schnitt}\}$$

entspricht, heißt ein maximaler Fluss.

Beispiel 4.3 *Bezug ist wieder Beispiel 4.1 (Seite 96). Wenn $A(X, \overline{X})$ ein Schnitt in diesem Graphen ist, dann muß $q \in X$ und $s \in \overline{X}$ sein, und jeder der vier inneren Ecken v_2, v_3, v_4, v_5 kann entweder in X oder in $\overline{X}$ enthalten sein. Es folgt, dass es $2^4 = 16$ mögliche Schnitte existieren*[1]*. Wenn die Kapazität jeder dieser Schnitte betrachtet wird, ergibt sich u.a. dass der Schnitt $A(\{q = v_1, v_2, v_3, v_4\}, \{v_5, s = v_6\})$ die kleinstmögliche Kapazität aller Schnitte von 4 hat. Daher folgt, dass jeder Fluss des Graphen einen Wert von höchstens 4 haben kann.*

In dem nächsten Abschnitt wird eine Methode vorgestellt, um einen maximalen Fluss zu erzeugen, wobei der gegebene Fluss schrittweise erhöht wird.

4.1.2 Vergrößernde Wege

Das MaximalFlussproblem ist ein lineares Optimierungsproblem, aber zu seiner Lösung wird nicht das Simplex-Verfahren[2] (oder ein ähnlich allgemeingültiges Verfahren), sondern ein der speziellen Aufgabenstellung angepaßter Ansatz verwendet. Er basiert auf dem Begriff des vergrößernden Weges.

[1] Allgemeiner: Ein Graph G hat 2^n mögliche Schnitte, wenn es n innere Ecken in G gibt.

[2] Gegeben ist eine lineare Gleichung f, sowie ein System von Ungleichungen, die nähere Aussagen über die Unbekannten von f machen. Durch das Simplexverfahren ist es möglich, diese Unbekannten zu ermitteln, für den Fall dass f ein Maximum annimmt. Das Verfahren ähnelt dem Gauß'schen Eliminierungsverfahren, da es auf der Darstellung in Tabellen (mit Vertauschungen und Pivot) basiert.

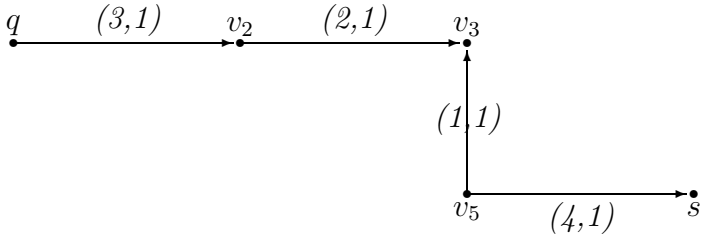
Definition 4.4 *Ein ungerichteter Weg von der Quelle q zur Senke s heißt ein* vergrößernder Weg*, wenn gilt:*

- *Für jede Kante e_{ij}, die auf dem Weg entsprechend ihrer Richtung durchlaufen wird (sie wird als* Vorwärtskante *bezeichnet), ist $f(e_{ij}) < c(e_{ij})$.*
- *Für jede Kante e_{ij}, die auf dem Weg entgegen ihrer Richtung durchlaufen wird (sie wird als* Rückwärtskante *bezeichnet), ist $f(e_{ij}) > 0$.*

Beispiel 4.4 *(Fortsetzung) Bezug ist wieder Beispiel 4.1 (Seite 96). In diesem Graphen ist*

$$q = v_1,\ v_2,\ v_3,\ v_5,\ s = v_6$$

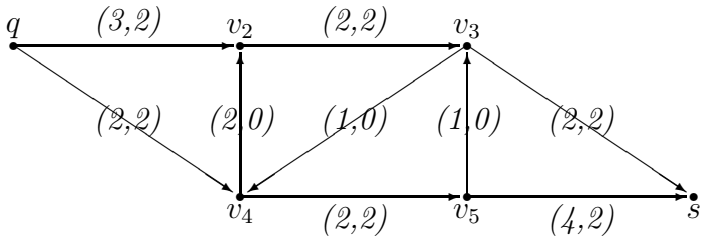
ein vergrößernder Weg:



Zu beachten ist, dass die Rückwärtskante von v_3 nach v_5 nur deshalb in den vergrößerten Weg aufgenommen wird, weil dort bereits ein Fluss besteht, der dann für eine Steigerung umgeleitet werden kann.

Mit Hilfe eines vergrößernden Weges läßt sich die Flussstärke d von q nach s folgendermaßen steigern:

Sei $\delta_s > 0$ die minimale Differenz zwischen $c(e_{ij})$ und $f(e_{ij})$ für alle Vorwärtskanten des Weges, sowie zwischen $f(e_{ij})$ und 0 für alle Rückwärtskanten des Weges, im Beispiel also $\delta_s = 1$. Dann wird $f(e_{ij})$ für jede Vorwärtskante auf dem Weg um δ_s erhöht und für jede Rückwärtskante um δ_s erniedrigt. Damit steigt der Fluss von q nach s um δ_s unter Einhaltung aller Nebenbedingungen:



Dazu nun die formalen Grundlagen.

Definition 4.5 *Ist eine Kantenfolge $W = v_1 v_2 \dots v_n$ in dem unterliegenden Graphen G' des Graphen*[3] *G gegeben, dann sind die zugehörigen Kanten in G entweder in der Form $e_{(i-1)i}$, genannt* Vorwärtskante *von W, oder $e_{i(i-1)}$, genannt* Rückwärtskante *von W.*

Wenn f ein Fluss in G ist, wird einer Kantenfolge W eine nichtnegative Zahl $i(W)$, das Inkrement[4] *von W, zugeordnet, wobei*

$$i(W) = \min\{i(e_{ij}) \| e_{ij} \textit{ist eine Kante der Kantenfolge} W\}$$

$$i(e_{ij}) = \begin{cases} c(e_{ij}) - f(e_{ij}) & \textit{falls } e_{ij} \textit{ eine Vorwärtskante von } W \\ f(e_{ij}) & \textit{falls } e_{ij} \textit{ eine Rückwärtskante von } W \end{cases}$$

Die nächste Definition bezeichnet die Kantenfolgen, die nicht ihre volle Kapazität ausnutzen.

Definition 4.6 *Die Kantenfolge W heißt* f-gesättigt*, wenn $i(W) = 0$, und* f-ungesättigt*, wenn $i(W) > 0$.*

4.1.3 Maximaler Fluss

Der zentrale Satz zur Bestimmung von Maximalflüssen lautet nun:

Satz 4.2 *Wenn in einem Graphen G ein Fluss der Stärke d von der Quelle q zur Senke s fließt, gilt genau eine der beiden Aussagen:*

1. *Es gibt einen vergrößernden Weg.*
2. *Es gibt einen Schnitt $A(X, \overline{X})$ mit $c(X, \overline{X}) = d$.*

Beweis: Da sich (1.) und (2.) per Definition gegenseitig ausschließen, ist also noch zu beweisen, dass stets mindestens eine von ihnen wahr ist.

Ausgehend von q versuchen wir, andere Ecken über ungerichtete Wege zu erreichen, wobei wir nur dann längs einer Kante von v_i nach v_j gehen, wenn $f(e_{ij}) < c(e_{ij})$ oder $f(e_{ji}) > 0$ gilt. Entweder können wir auf diese Weise die Ecke s erreichen, dann wurde ein vergrößernder Weg gefunden, oder wir bezeichnen mit Q alle Ecken, die von q aus erreichbar sind (einschließlich q selbst) und mit $S := V \setminus Q = \overline{Q}$ alle anderen Ecken. $A(Q, \overline{Q})$ sei der zugehörige Schnitt. Für jede von Q nach S gerichtete

[3] Zur Erinnerung, in diesem Abschnitt sind, sofern nichts anderes gesagt, alle Graphen schwach zusammenhängend, schlicht und gerichtet.

[4] Inkrement für den bestehenden Fluss; ggf. wird dies jedoch bei einer Reorganisation auch von einem bestehenden Fluss abgezogen.

Kante aus $A(Q, \overline{Q})$ muß dann gelten $f(e_{ij}) = c(e_{ij})$, und für jede von S nach Q gerichtete Kante aus $A(Q, \overline{Q})$ muß dann gelten $f(e_{ij}) = 0$, sonst wäre die in S gelegene Anfangs- bzw. Endecke von q aus über einen ungerichteten Weg der zuvor beschriebenen Art erreichbar. Nach Satz 4.1 (Seite 98) ist dann

$$d = f(Q, \overline{Q}) - f(\overline{Q}, Q) = c(Q, \overline{Q}) - 0 = c(Q, \overline{Q})$$

q.e.d.

(1.) beschreibt die Erweiterbarkeit, d.h. ist der maximale Fluss noch nicht gefunden, kann noch ein vergrößender Weg und damit eine Steigerung des aktuellen Flusses gefunden werden. (2.) beschreibt das Ende: der „Flaschenhals“, d.h. der minimale Schnitt, ist gefunden und damit ist der maximal mögliche Fluss gefunden.

Da im Fall der Gültigkeit von (2.) nicht nur d seinen Maximalwert, sondern auch $c(X, \overline{X})$ seinen Minimalwert annimmt, wird der vorstehende Satz meist in der folgenden Form zitiert.

Satz 4.3 (Max-flow-min-cut Theorem von Ford und Fulkerson)
In einem schwach zusammenhängendem schlichten Digraphen G mit genau einer Quelle q und genau einer Senke s sowie der Kapazitätsfunktion c und dem Fluss f ist das Minimum der Kapazität eines q und s trennenden Schnitts gleich der Stärke eines maximalen Flusses von q nach s.

Der Algorithmus von Ford und Fulkerson baut im wesentlichen auf dem Beweis zu diesem Satz auf. In diesem Buch soll jedoch nur der Algorithmus vorgestellt werden.

Bemerkung 4.1 (Ganzzahligkeitseigenschaft) *Bei ganzzahligen Werten $f(e_{ij})$ und $c(e_{ij})$ muß die zu einem vergrößernden Weg gehörende Größe δ_s ebenfalls ganzzahlig sein. Wenn wir also bei ganzzahligen Kapazitäten $c(e_{ij})$ ausgehend von $f(e_{ij}) = 0$ für alle $i, j = 1, \ldots, n$ den Maximalwert für d durch wiederholtes Suchen vergrößernder Wege bestimmen, werden alle dabei auftretende Werte für $f(e_{ij})$ und d ganzzahlig sein.*

Bei rationalen Werten der $c(e_{ij})$ heißt dies, dass die ausgehend von $f(e_{ij}) = 0$ mittels vergrößernder Wege bestimmten Werte für $f(e_{ij})$ nie einen größeren Hauptnenner haben können als die Werte $c(e_{ij})$.

4.1.4 Der Algorithmus von Ford und Fulkerson

Die bisherigen Überlegungen sollen jetzt als Algorithmus formuliert werden. Den Ecken v_i des Graphen werden während der Suche nach einem vergrößernden Weg Marken $(Vorg_i, \delta_i)$ zugewiesen. Dabei gibt $Vorg_i$ den Vorgänger von v_i auf einem vergrößernden Weg und δ_i die bisher auf diesem Weg maximal mögliche Änderung der Flussstärke an. $Vorg_i$ ist dabei mit einem Vorzeichen versehen, das angibt, ob von $Vorg_i$ nach v_i eine Kante entsprechend ihrer Richtung (+) oder entgegen ihrer Richtung (–) durchlaufen wird[5].

Algorithmus 4.1 *Gegeben sei ein schwach zusammenhängender, schlichter Digraph $G = (V, E)$, eine Kapazitätsfunktion c und ein Fluss f.*

1. *(Initialisierung)*
   Weise allen Kanten $f(e_{ij})$ als einen (initialen) Wert zu, der die Nebenbedingungen[6] erfüllt. Markiere q mit (undefiniert, ∞).

2. *(Inspektion und Markierung)*

   (a) *Falls alle markierten Ecken inspiziert wurden, gehe nach 4.*

   (b) *Wähle eine beliebige[7] markierte, aber noch nicht inspizierte Ecke v_i und inspiziere sie wie folgt (Berechnung des Inkrements)*

   - *(Vorwärtskante) Für jede Kante $e_{ij} \in O(v_i)$ mit unmarkierter Ecke v_j und $f(e_{ij}) < c(e_{ij})$ markiere[8] v_j mit $(+v_i, \delta_j)$, wobei δ_j die kleinere der beiden Zahlen $c(e_{ij}) - f(e_{ij})$ und δ_i ist.*

[5] In anderen Quellen wird z.B. ein Residualnetzwerk verwendet. Dieses übernimmt die Rolle der Markierungen in der hier für dieses Buch gewählten Beschreibung. Diese Methode ist *eager*, da sie zunächst alle Möglichkeiten der Veränderungen berechnet (dargestellt im Residualnetzwerk) und sucht dann auf diesem Netzwerk einen Weg (über Vorwärtskanten). Die hier im Buch beschriebene Methode ist *lazy*, da sie die Veränderungsmöglichkeiten erst berechnet, wenn sie an der Wegefindung direkt beteiligt werden. Effizienter Implementieren lässt sich die Variante mit dem Residualnetzwerk.
Im Allgemeinen sollte man bei unterschiedlichen Beschreibungen des gleichen Algorithmus Zuordnungen vornehmen, um die unterschiedlichen Strategien zu erkennen und das Kernkonzept der Algorithmen identifizieren, denn schlieSSlich erfüllen sie die gleiche Aufgabe! Wird beides vermischt oder z.B. das Kernkonzept des Algorithmus durch Vermischung versehentlich wegimplementiert, arbeitet der Algorithmus u.U. nicht korrekt oder sehr ineffizient!

[6] Sofern $f(e_{ij}) > 0$ gewählt wird, bezieht sich dies auf die Machbarkeit des initialen Flusses: er muß von q zu s durchfliessen.

[7] Ford Fulkerson sucht durch das „beliebig“ einen *beliebigen* vergrößernden Weg; Edmond und Karp suchen hier den *kürzesten* vergrößernden Weg, der an dieser Stelle durch Verwendung einer Queue für die markierten Ecken erzielt werden kann.

[8] Fluss-Inkrement als Inkrement auf den aktuellen Fluss.

- *(Rückwärtskante) Für jede Kante $e_{ji} \in I(v_i)$ mit unmarkierter Ecke v_j und $f(e_{ji}) > 0$ markiere*[9] *v_j mit $(-v_i, \delta_j)$, wobei δ_j die kleinere der beiden Zahlen $f(e_{ji})$ und δ_i ist.*

(c) Falls s markiert ist, gehe zu 3., sonst zu 2.(a).

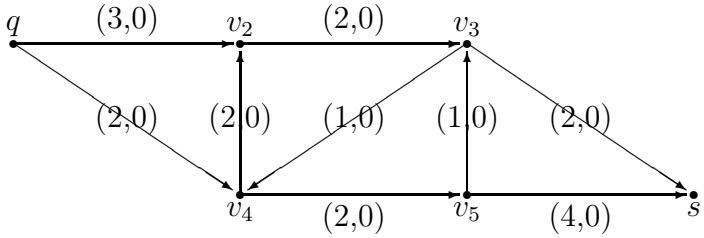
3. *(Vergrößerung der Flussstärke)*
   Bei s beginnend lässt sich anhand der Markierungen der gefundene vergrößernde Weg bis zur Ecke q rückwärts durchlaufen. Für jede Vorwärtskante wird $f(e_{ij})$ um δ_s erhöht[10]*, und für jede Rückwärtskante wird $f(e_{ji})$ um δ_s vermindert. Anschließend werden bei allen Ecken mit Ausnahme von q die Markierungen entfernt. Gehe zu 2.*

4. *Es gibt keinen vergrößernden Weg. Der jetzige Wert von d ist optimal. Ein Schnitt $A(X, \overline{X})$ mit $c(X, \overline{X}) = d$ wird gebildet von genau denjenigen Kanten, bei denen entweder die Anfangsecke oder die Endecke inspiziert ist.*

In Schritt 1. wird i.allg. $f(e_{ij}) := 0$ für alle i und j gewählt.

4.1.5 Beispiel

Der Algorithmus wird auf folgenden gegebenen Graphen angewendet.



Die in den vorangegangenen Beispielen dargestellte Situation, insbesondere die $(1, 1)$ Markierung der Kante (v_5, v_3) läßt sich so durch den Algorithmus nicht verwenden.

Nach der Initialisierung gilt folgende Tabelle, wobei inspizierte Ecken mit * markiert werden, nur markierte, aber noch nicht inspizierte Ecken stehen in der Tabelle ohne * und nicht markierte Ecken sind dort nicht aufgeführt:

gekennzeichnete Ecke	q
Kennzeichnung	(undefiniert, ∞)

[9] Fluss-Inkrement als Dekrement auf den aktuellen Fluss im Sinne einer Reorganisation.

[10] δ_s ist das minimale Inkrement, um dass die Kanten des vergrößernden Weges modifiziert werden.

Gewählt wird in Schritt 2. die Ecke q; andere stehen nicht zur Verfügung.

gekennzeichnete Ecke	q*	v_2	v_4
Kennzeichnung	(undefiniert, ∞)	(+q,3)	(+q,2)

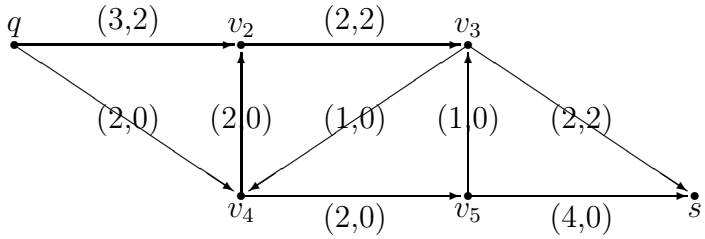
Gewählt wurde (beliebig) v_2.

gekennzeichnete Ecke	q*	v_2*	v_4	v_3
Kennzeichnung	(undefiniert, ∞)	(+q,3)	(+q,2)	(+v_2,2)

Gewählt wurde (beliebig) v_3.

gekennzeichnete Ecke	q*	v_2*	v_4	v_3*	s
Kennzeichnung	(undefiniert, ∞)	(+q,3)	(+q,2)	(+v_2,2)	(+v_3,2)

Nun wurde die Senke s markiert und der Algorithmus verzweigt zu Schritt 3. Dort wird der gefundene vergrößender Weg $W = \{q, v_2, v_3, s\}$ um das Inkrement $i(W) = 2$ erhöht. Resultat ist:



Eine neue Runde beginnt. Als erstes wird wieder q ausgewählt, da keine weiteren Ecken zur Verfügung stehen.

gekennzeichnete Ecke	q*	v_2	v_4
Kennzeichnung	(undefiniert, ∞)	(+q,1)	(+q,2)

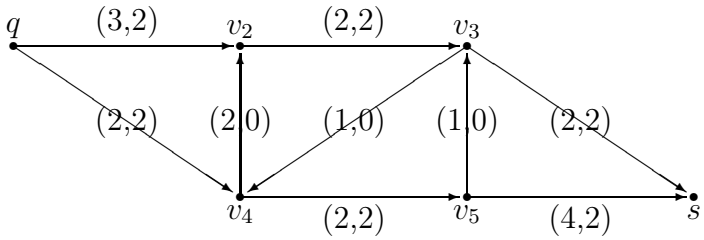
Gewählt wurde (beliebig) v_2. Keine neuen Ecken können jedoch markiert werden. Gewählt wird nun v_4.

gekennzeichnete Ecke	q*	v_2*	v_4*	v_5
Kennzeichnung	(undefiniert, ∞)	(+q,1)	(+q,2)	(+v_4,2)

Gewählt wurde v_5.

gekennzeichnete Ecke	q*	v_2*	v_4*	v_5*	v_3	s
Kennzeichnung	(undefiniert, ∞)	(+q,1)	(+q,2)	(+v_4,2)	(+v_5,1)	(+v_5,2)

Nun wurde erneut die Senke s markiert und der Algorithmus verzweigt zu Schritt 3. Dort wird der gefundene vergrößender Weg $W = \{q, v_4, v_5, s\}$ um das Inkrement $i(W) = 2$ erhöht. Resultat ist:



Bei der nächsten Runde wird zunächst v_2 markiert werden können, dann jedoch werden keine weiteren Ecken mehr markiert. Der Fall 2.(a) tritt ein, d.h. der Algorithmus verzweigt zu Schritt 4. Eine Zerlegung kann nun wie folgt aussehen: $Q = \{q, v_2, \}$ und $\overline{Q} = \{v_3, v_4, v_5, s\}$. Es gilt $f(Q, \overline{Q}) - f(\overline{Q}, Q) = 4 - 0 = 4$. Somit war der Fluss mit der Stärke 3 im Beispiel 4.1 (Seite 96) nicht optimal.

4.1.6 Überlegungen zur Komplexität

Zum Finden eines vergrößernden Weges muß schlimmstenfalls jede Kante zweimal inspiziert werden (in jeder Richtung einmal), wobei jede Inspektion eine konstante (d.h. $O(1)$) Anzahl von Arbeitsschritten umfaßt (Vergleich von $f(e_{ij})$ mit $c(e_{ij})$, bzw. mit 0, Änderung der Markierung einer Ecke). Nach jeweils $O(|E|)$ Schritten ist also entweder ein neuer vergrößernder Weg gefunden oder nachgewiesen, dass es keinen mehr gibt.

Wenn d_{max} der Maximalwert von d ist, kann es allerdings sein, dass d_{max} vergrößernde Wege gefunden werden müssen, bis d sein Maximum auch tatsächlich annimmt. Die Komplexität des Algorithmus muß also mit $O(|E| \cdot d_{max})$ angegeben werden.

Die Dateneingabe für das MaximalFlussproblem kann als lexikographisch geordnete Liste der Werte $c(e_{ij})$ für alle Eckenpaare (v_i, v_j) (mit $c(e_{ij}) = 0$ für nichtexistierende Kanten) erfolgen. Bezeichnen wir den Maximalwert der Kapazitäten mit c_{max} so läßt sich der Speicherplatzbedarf der Eingabe abschätzen durch $O(|V|^2 \cdot \log(c_{max}))$. Da Kantenkapazitäten, die größer sind als d_{max}, ohnehin keinen EinFluss auf die Lösung der Aufgabe haben, können wir

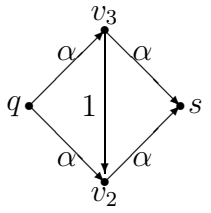
$c_{max} \leq d_{max}$ voraussetzen. (Dies stellt die ungünstigste Situation für die Beziehung zwischen Eingabeumfang und Arbeitsaufwand dar.) Dann sehen wir, dass der Arbeitsaufwand (wegen $O(c_{max}) = O(\mathrm{e}^{\log(c_{max})})$) mit zunehmendem Eingabeumfang exponentiell ansteigt. Der Algorithmus ist also nicht polynomial.

Eine Abänderung des Algorithmus schafft jedoch Abhilfe:

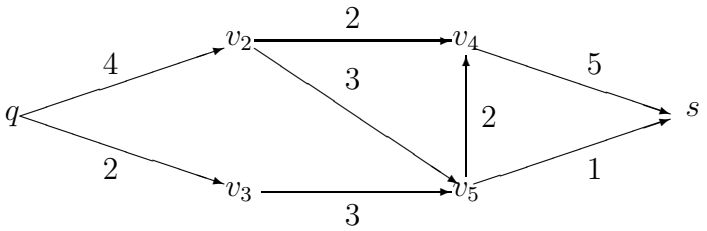
Satz 4.4 *Wenn jede Vergrößerung der Flussstärke d durch einen vergrößernden Weg minimaler Kantenanzahl erfolgt, dann sind höchstens $O(|E| \cdot |V|)$ vergrößernde Wege zu berechnen, bis d seinen Maximalwert erreicht hat.*

Aufgaben

1. In dem folgenden Graphen sind die Kapazitäten der Kanten angegeben. Dabei sei α eine große natürliche Zahl. Zeigen Sie, wie die Wahl der vergrößernden Wege die Anzahl der Arbeitsschritte beeinFlusst, die nötig sind, damit ausgehend von $d = 0$ die Stärke des Flusses von q nach s ihren Maximalwert annimmt:



2. Berechnen Sie die Stärke des Maximalflusses von q nach s in dem folgenden Graphen. Spielen Sie die verschiedenen Situationen durch, die sich aus unterschiedlichen Inspektionsreihenfolgen der Ecken ergeben (z.B. immer v_2 vor v_3 inspizieren oder immer v_3 vor v_2 oder beim ersten vergrößernden Weg v_2 vor v_3 und dann umgekehrt usw.).



3. Betrachten Sie den Graphen in Abbildung 4.1 (Seite 107). Die Bewertungen der Kanten geben die Kapazität und den Fluss an.

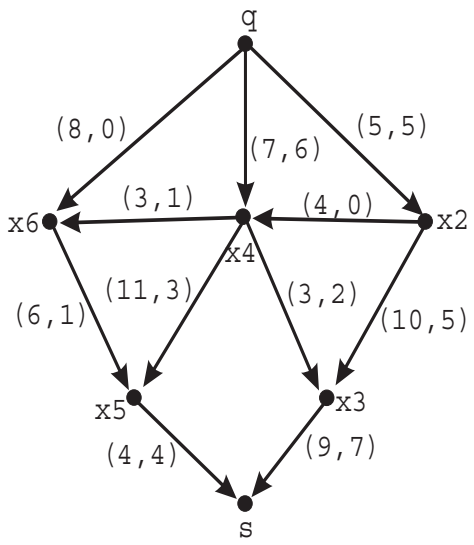


Abbildung 4.1: Flussproblem

(a) Bestimmen Sie den Wert des aktuellen Flusses in dem Graphen. Erklären Sie kurz, wie Sie die Berechnung vornehmen.

(b) Finden Sie einen vergrößernden Weg und bestimmen Sie ggf. den erhöhten Fluss mittels dem Algorithmus von Ford und Fulkerson. Fertigen Sie eine Dokumentation an, aus der der Ablauf deutlich wird.

(c) Bestimmen Sie einen minimalen Schnitt. Erklären Sie, wie Sie die Bestimmung vornehmen.

(d) Geben Sie einen maximalen Fluss für diesen Graphen an. Erklären Sie kurz, wie Sie die Berechnung vornehmen.

4. **Minimaler Schnitt** (? Punkte)
Betrachten Sie den Graphen in Abbildung 4.2 (Seite 108). Die Bewertungen der Kanten geben die Kapazität an.

(a) Bestimmen Sie den maximalen Fluss mittels dem Algorithmus von Ford und Fulkerson. Fertigen Sie eine Dokumentation an, aus der der Ablauf deutlich wird.

(b) Bestimmen Sie einen minimalen Schnitt. Erklären Sie, wie Sie die Bestimmung vornehmen. Berechnen Sie den maximalen Fluss am minimalen Schnitt.

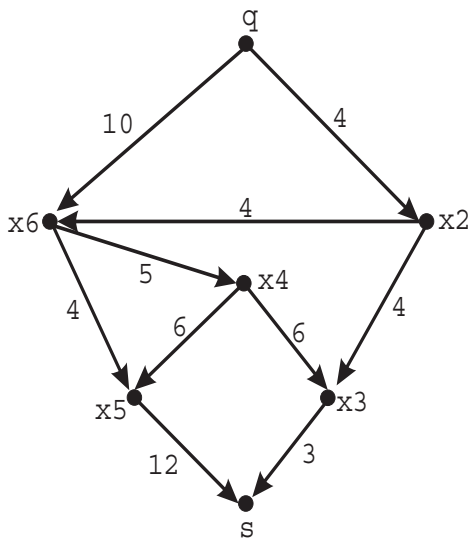


Abbildung 4.2: Flussnetzwerk

5. **Maximaler Fluss** (?? Punkte)
   Führen Sie den Algorithmus von Ford und Fulkerson auf dem Graphen in Abbildung 4.3 (Seite 109) durch (bis zum Abbruch!). Die Markierungen sind in der Form `(Kapazität, Fluss)`. Fertigen Sie eine Dokumentation an, aus der der Ablauf des Algorithmus deutlich wird. Bestimmen Sie den maximalen Fluss und einen minimalen Schnitt!

   Zeichnen Sie zu dem Flussnetzwerk nach dem dritten vergrößernden Schritt das zugehörige Residualnetzwerk.

6. Es sei ein Flussnetzwerk $F = (V, E, f, c)$ mit einem schwach zusammenhängenden, schlichtem, gerichteten Graphen $G = (V, E)$, dem zugehörigen Fluss f und der zugehörigen Kapazität c gegeben.

   Wir definieren ein **Residualnetzwerk** $R_F := (V', E', r)$ des Flussnetzwerkes $F = (V, E, f, c)$ wie folgt:

   - $V' = V$, d.h. die Menge der Ecken ist gleich.
   - Die Residualfunktion r ordnet jeder Kante $e'_{ij} \in E'$ eine positive rationale Zahl (> 0) zu.
   - $\forall (v_i, v_j) = e_{ij} \in E$ ist
     - $e'_{ij} \in E'$, falls $c(e_{ij}) - f(e_{ij}) > 0$, mit $r(e'_{ij}) := c(e_{ij}) - f(e_{ij})$
     - $e'_{ji} \in E'$, falls $f(e_{ij}) > 0$, mit $r(e'_{ij}) := f(e_{ij})$.

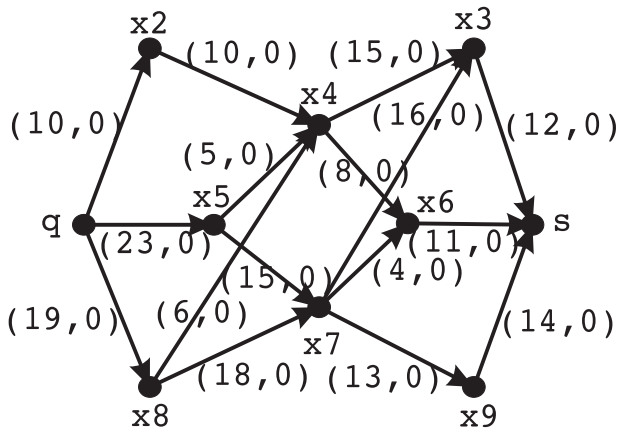


Abbildung 4.3: Flussproblem

Zeichnen Sie zu dem Flussnetzwerk aus Abbildung 4.1 (Seite 107) das zugehörige Residualnetzwerk, in dem die Kanten e'_{ij} mit dem Wert $r(e'_{ij})$ beschriftet werden.

4.2 Andere Flussprobleme

Das MaximalFlussproblem ist nur das einfachste Beispiel einer Fragestellung über Flüsse auf einem gerichteten Graphen. Beispiele für wichtige Abwandlungen dieser Aufgabenstellung sind:

- Für den Transport einer Mengeneinheit längs einer Kante fallen (je nach Kante unterschiedliche) Kosten an. Wie kann ein Fluss vorgegebener Stärke kostenminimal von q nach s geleitet werden?

- Für jede Kante ist zusätzlich zu ihrer Kapazität $c(e_{ij})$ noch eine Mindestauslastung $l(e_{ij})$ vorgeschrieben. Gibt es überhaupt einen Fluss von q nach s, so dass auf jeder Kante $l(e_{ij}) \leq f(e_{ij}) \leq c(e_{ij})$ gilt?

- In dem Graphen $G(V, E)$ gibt es mehrere Quellen $q_1, \ldots, q_m$ und Senken $s_1, \ldots, s_m$. Zwischen jeweils übereinstimmend indizierten Quellen und Senken q_i, s_i soll eine bestimmte Menge d_i eines Gutes g_i fließen. Ist es möglich, alle diese Gütermengen gleichzeitig fließen zu lassen, ohne dass die Kapazität irgendeiner Kante überschritten wird?

4.3 Zuordnungs- und Transportprobleme

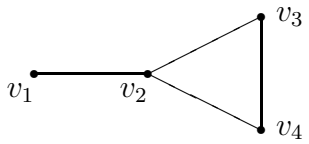
In den Kontext der Flußprobleme gehören auch einige andere häufig auftretende Probleme.

4.3.1 Matchings in einem bipartiten Graphen

Beispiel 4.5 *In einer Firma stehen zur Übernahme von n Jobs $j_1, \ldots, j_n$, die jeweils einen ganzen Tag Arbeitszeit benötigen, m Arbeiter $w_1, \ldots, w_m$ zur Verfügung. Jeder Arbeiter kann aber aufgrund seiner jeweiligen Qualifikation nur bestimmte der Arbeiten ausführen. Welche Arbeit sollen welchem Arbeiter übertragen werden, damit während eines Tages möglichst viele Arbeiten erledigt werden (und möglichst wenige Arbeiter ohne Arbeit sind)?*

Definition 4.7 (Matching) *Sei $G(V, E)$ ein ungerichteter Graph. Eine Menge $F \subseteq E$ heißt ein* Matching *(oder: eine Zuordnung), wenn keine zwei Elemente von F eine Ecke gemeinsam haben.*

Beispiel 4.6 *In dem folgenden Graphen ist $F := \{v_1v_2, v_3v_4\}$ ein Matching*[11].



Beispiel 4.7 *(Fortsetzung) Das Problem der Zuordnung von Arbeitern zu Arbeiten kann folgendermaßen durch einen Graphen dargestellt werden: Für jeden Arbeiter und jede Arbeit wird je eine Ecke eingeführt. Zwei Ecken sind genau dann durch eine Kante verbunden, wenn sie einen Arbeiter und eine von ihm ausführbare Arbeit darstellen. Der entstehende Graph ist also bipartit. Da jedem Arbeiter nur höchstens eine Arbeit zugeordnet werden kann, wird somit nach einem Matching gesucht, das aus möglichst vielen Kanten besteht.*

Unter dem Problem des maximalen Matchings auf bipartiten Graphen (meist vereinfachend maximales bipartites Matchingproblem genannt) verstehen wir folgende Aufgabenstellung:

[11] Dieser Graph ist nicht als bipartiter Graph darstellbar, da er einen Kreis mit ungerader Anzahl an Ecken beinhaltet!

Maximales bipartites Matchingproblem

In einem bipartiten ungerichteten Graphen $G(X \cup Y, E)$ soll ein Matching F so bestimmt werden, dass $|F|$ maximal wird.

Definition 4.8 *Sei F ein Matching in einem Graphen $G(V, E)$.*

1. *Eine Ecke, die mit keiner Kante aus F inzident ist, heißt* von F unversorgt.
2. *Ein Weg in G, der abwechselnd Kanten aus F und aus $E \setminus F$ enthält, heißt ein* alternierender Weg.
3. *Ein alternierender Weg, dessen erste und letzte Ecke unversorgt sind, heißt ein* vergrößernder Weg.
4. *F heißt* perfekt*, wenn G keine unversorgten Ecken besitzt.*

Mit Hilfe eines vergrößernden Weges kann ein Matching F' mit einer gegenüber F um eins erhöhten Kantenanzahl gefunden werden: Sei W die Kantenmenge des vergrößernden Weges. Wenn aus F alle Kanten aus $W \cap F$ entfernt werden, sind sämtliche Ecken des vergrößernden Weges unversorgt. Deshalb entsteht durch Hinzufügen der Kantenmenge $W \setminus F$ wieder ein Matching. Das Matching $F' := (F \setminus (W \cap F)) \cup (W \setminus F)$ besitzt aber eine Kante mehr (denn es gibt zwei unversorgte Ecken weniger) als F.

Allgemein gilt:

Satz 4.5 *In einem Graphen $G(V, E)$ besitzt ein Matching F genau dann die maximal mögliche Anzahl von Kanten, wenn es keinen vergrößernden Weg gibt.*

Beim Problem des maximalen Flußes haben wir auch mit vergrößernden Wegen gearbeitet. Vergrößernde Wege dort wie auch hier zeigen eine mögliche Reorganisation (des Flußes/des Matchings) auf, die eine Verbesserung bzgl. des gestellten Problems erlaubt. Dies ist notwendig, da - im Unterschied zu z.B. dem Dijkstra-Algorithmus - bei der (lokalen) Festlegung auf einen Fluß bzw. ein Matching nicht garantiert werden kann, das die getroffene Entscheidung aus globaler Sicht optimal ist.

In einem bipartiten Graphen $G(X \cup Y, E)$ kann man auf folgende Weise stets einen alternierenden Weg finden, wenn es ihn gibt. Dieser Algorithmus wird als Ungarische Methode bezeichnet[12], da er auf die Arbeiten der ungarischen Mathematiker *König* und *Egerváry* zurückgeht. Er verfolgt im Sinne einer Breitensuche mehrere Versuche, einen vergrößernden Weg zu finden.

[12] Wird auch als Kuhn-Munkres-Algorithmus bezeichnet.

Ungarische Methode

Algorithmus 4.2 *In G liege ein Matching F vor (z.B. $F = \emptyset$).*

1. *Markiere alle unversorgten Ecken aus X. Alle anderen Ecken sind unmarkiert. Weiter mit 2.*

2. *Markiere in Y alle bislang unmarkierten Ecken, die mit einer im letzten Schritt markierten Ecke durch eine Kante aus $E \setminus F$ verbunden sind*[13]*, mit der Nummer dieser X-Ecke*[14]*.*

   (a) *Wurde keine Ecke in Y neu markiert, dann gibt es keinen vergrößernden Weg. F ist maximal. ENDE*

   (b) *Wurde eine unversorgte Ecke in Y neu markiert, dann wurde ein vergrößernder Weg gefunden. Weiter mit 4.*

   (c) *Sonst (d.h. nur versorgte Ecken in Y wurden markiert) weiter mit 3.*

3. *Markiere in X alle Ecken, die mit den im letzten Schritt neu markierten Ecken aus Y durch eine Kante aus F verbunden sind. Weiter mit 2.*

4. *Ausgehend von der beim letzten Durchlauf von Schritt 2. markierten unversorgten Y-Ecke verfolge den vergrößernden Weg W zurück, indem jeweils von einer Y-Ecke zu der aus der Markierung ersichtlichen X-Ecke und von einer X-Ecke zu der über eine Kante aus F erreichbaren Y-Ecke gegangen wird. Bilde hieraus ein Matching $F' := (F \setminus (W \cap F)) \cup (W \setminus F)$ mit $|F'| = |F| + 1$. Lösche alle Markierungen. Weiter mit 1.*

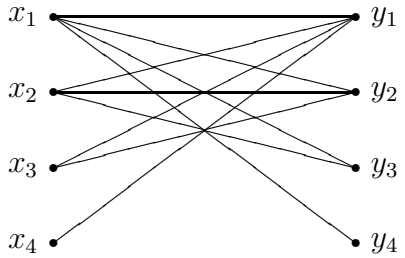
Konvention: Algorithmen bieten manchmal Wahlmöglichkeiten an, die zu unterschiedlichen Ergebnissen in dem Sinne führen können, dass auf der einen Seite das Optimum erreicht wird, auf der anderen Seite jedoch unterschiedliche Akteure daran beteiligt sind. Daher spielt für diese Algorithmen diese Wahlmöglichkeit keine Rolle: ein optimales Ergebnis wird so oder so gefunden. In einer konkreten Implementierung bzw. manuellen Berechnung muß jedoch diese Wahl entschieden werden. Dies geschieht manchmal indirekt (und leider auch unbewusst) durch die enstprechende ADT oder explizit durch ein Wahlkriterium. Bei dem aktuell betrachteten Algorithmus ist die „gleichzeitige Bearbeitung“ mehrerer Ecken eine solche, für das Endergebnis irrelevante

[13] Eine Kante aus F ist unmöglich, da dann die zugehörige Ecke aus X schon versorgt wäre!

[14] Breitensuche!

Wahlmöglichkeit. Für die folgende manuelle Berechnung wird diese so umgesetzt, dass die betreffenden Ecken in ihrer lexikographischen Reihenfolge nacheinander bearbeitet werden.

Beispiel 4.8 *Mit Hilfe der Ungarischen Methode wird ein perfektes Matching für den folgenden bipartiten Graphen mit 8 Ecken bestimmt.*



Wir beginnen mit dem Matching $F = \emptyset$. Beim ersten Erreichen von Schritt 4 des Algorithmus wird $F := \{x_1y_1\}$ gesetzt, unter der Voraussetzung, dass in Schritt 2. die Ecke y_1 mit 1 markiert wurde und in Schritt 4. diese Ecke gewählt wurde. Beim zweitenmal wird x_2y_2 zu F hinzugefügt. Auch hier vorausgesetzt, dass in Schritt 2. y_2 mit 2 markiert wurde und in Schritt 4. diese Ecke gewählt wurde. Beim drittenmal liegt ein alternierender Weg mit drei Kanten vor, und wir erhalten $F := \{x_1y_3, x_3y_1, x_2y_2\}$. Hier vorausgesetzt, dass y_3 mit 1 markiert wurde und y_1 mit 3. Dieser alternierende Weg (über x_1y_1) wird nun in dem Matching aufgenommen; x_1y_1 wird daher entfernt. Beim viertenmal schließlich ergibt sich das perfekte Matching $F := \{x_1y_4, x_2y_3, x_3y_2, x_4y_1\}$. Hier wurden (zwingend) folgende Markierungen vorgenommen: y_1 mit 4, y_2 mit 3, y_3 mit 2 und y_4 mit 1. Da nun ein alternierender Weg gefunden wurde, der über alle Kanten aus dem aktuellen F läuft, werden diese durch das neue Matching ersetzt.

Komplexität

Als Eingabe benötigt der Algorithmus $|X| \cdot |Y|$ Angaben über die Existenz, bzw. Nichtexistenz von Kanten.

Teil 1. des Algorithmus kann höchstens $\min\{|X|, |Y|\}$-mal aufgerufen werden, weil danach in der kleineren der beiden Eckenteilmengen keine unversorgte Ecke mehr existiert.

Zwischen zwei Aufrufen von 1. kann höchstens folgender Arbeitsaufwand anfallen:

1. benötigt $O(|X|)$ Schritte.
2. benötigt $O(|E|)$ Schritte.
3. benötigt $O(|F|)$ Schritte.
4. benötigt $O(|F|)$ Schritte.

Damit beträgt die Gesamtzahl der auszuführenden Arbeitsschritte maximal

$$\begin{aligned} & O(\min\{|X|,|Y|\} \cdot (O(|X|) + O(|E|) + 2 \cdot O(|F|))) \\ = \; & O(\min\{|X|,|Y|\} \cdot O(|E|)) \\ = \; & O(\max\{|X|,|Y|\}^3) \end{aligned}$$

Die Ungarische Methode ist also polynomial.

Finden einer Anfangslösung mit dem Greedy-Algorithmus

Der Greedy-Algorithmus kann für das maximale bipartite Matchingproblem so formuliert werden:

– Bringe die Kanten des vorgelegten Graphen in eine beliebige Reihenfolge (z.B. die lexikographische Ordnung der zugehörigen Eckenpaare).
– Beginnend mit $F := \emptyset$ gehe die Kanten der Reihe nach durch und füge nur dann eine Kante nicht zu F hinzu, wenn sie mit einer Kante aus F eine gemeinsame Ecke hat.

Das Matching, das man auf diese Weise erhält, ist zwar nicht unbedingt maximal. Wenn man es aber als Startwert für die Ungarische Methode verwendet, werden gegenüber dem Startwert $F := \emptyset$ einige Durchläufe mit entsprechendem Markierungsaufwand eingespart. Dabei ist für den Greedy-Algorithmus eine gewisse „Mindestqualität“ garantiert:

Satz 4.6 *Jedes vom Greedy-Algorithmus gelieferte Matching besitzt mindestens halb soviele Kanten wie ein maximales Matching.*

Beweis: Ein maximales Matching habe $|F_{\max}|$ Kanten, während der Greedy-Algorithmus ein Matching mit r Kanten geliefert habe. Die mit den Kanten des Greedy-Matchings inzidenten Ecken bilden aber eine Eckenüberdeckung des Graphen (falls eine Kante mit keiner dieser Ecken inzident wäre, könnte sie zum Greedy-Matching hinzugefügt werden) mit $2r$ Ecken. Wegen der Dualität der beiden Probleme kann aber kein Matching mehr Kanten enthalten als eine beliebige Eckenüberdeckung desselben Graphen an Ecken enthält. Folglich ist

$$r \leq |F_{\max}| \leq 2r$$

Durch Vorschalten des Greedy-Algorithmus kann man also stets mindestens die Hälfte aller Iterationen der Ungarischen Methode einsparen (allerdings sind die verbleibenden Iterationen die aufwendigeren).

Der Heiratssatz

Die Frage, ob ein vorgelegter bipartiter Graph $G(X \cup Y, E)$ mit $|X| = |Y|$ ein perfektes Matching besitzt, kann auch ohne Verwendung der Ungarischen Methode beantwortet werden[15]. Es gilt nämlich der folgende *Satz von Hall*.

Satz 4.7 (Heiratssatz) *Für eine beliebige Menge Z von Ecken eines Graphen bezeichne $N(Z)$ die Nachbarschaft von Z, d.h. die Menge der mit Elementen aus Z adjazenten Ecken. Ein bipartiter Graph $G(X \cup Y, E)$ mit $|X| = |Y|$ besitzt genau dann ein perfektes Matching, wenn für jede Teilmenge $Z \subseteq X$ von X gilt: $|N(Z)| \geq |Z|$ (wenn also jede Teilmenge von X mindestens soviele Nachbarn besitzt wie sie selbst an Elementen umfaßt).*

Der Name dieses Satzes stammt von der folgenden „praktischen“ Anwendung des Zuordnungsproblems:

In einer Gruppe von Jugendlichen, die gleichviele Jungen und Mädchen umfaßt, sind etliche Mitglieder jeweils unterschiedlichen Geschlechts einander so sehr zugetan, dass sie bereit wären, einander zu heiraten. Ist es möglich, alle Mitglieder paarweise so miteinander zu verheiraten, dass jedes Mitglied einen seiner Wunschpartner bekommt?

Der Heiratssatz sagt dann aus, dass dies genau dann möglich ist, wenn für jede Gruppe Z von Mädchen die Gruppe $N(Z)$ derjenigen Jungen, die mindestens eines dieser Mädchen zu heiraten bereit ist[16], mindestens soviele Mitglieder hat wie Z.

Schreibweise als Maximalflußproblem

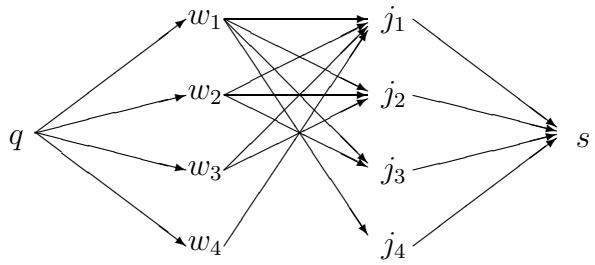
Das Zuordnungsproblem kann auch als Maximalflußproblem aufgefaßt werden: In der Darstellung des Problems als ungerichteter Graph werden alle Kanten von den Ecken w_i zu den Kanten j_k gerichtet. Außerdem verläuft je eine Kante von der Quelle q zu jeder der Ecken w_i und je eine Kante von jeder der Ecken j_k zu der Senke s. Alle Kanten haben die Kapazität 1. Gesucht ist ein Fluß maximaler Stärke von q nach s.

Ein vergößender Weg hier wäre dann auch bei der ungarischen Methode ein vergrößender Weg.

Beispiel 4.9 *Der Graph in Beispiel 4.8 wird so zu:*

[15] Allerdings ist der Satz nur für Graphen mit wenigen Ecken praktikabel bzw. wenn für ein Ergebnis nur wenige Ecken betrachtet werden müssen.

[16] Die Rede ist, wie gesagt, von einer Bereitschaft auf Gegenseitigkeit.



4.3.2 Andere Matchingprobleme

Optimales bipartites Matchingproblem

Unser obiges Beispiel für das Zuordnungsproblem kann dadurch weiter ausgebaut werden, dass für jeden Arbeiter nicht nur angegeben wird, ob er in der Lage ist, eine bestimmte Arbeit zu übernehmen, sondern diese Fähigkeit auch noch durch eine positive Zahl bewertet wird. Gesucht ist dann in der Darstellung als bipartiter Graph, bei dem die Kanten entsprechend den Fähigkeiten der Arbeiter zur Übernahme der jeweiligen Arbeit bewertet sind, ein Matching, bei dem die Summe der Kantenbewertungen maximal wird. Ein solches Matching (unabhängig von der Anzahl der Kanten!) wird optimales Matching genannt.

Die Suche nach einem optimalen Matching in einem bipartiten Graphen basiert auf der Schreibweise dieses Problems als linearem Optimierungsproblem. Aufgrund von Dualitätsüberlegungen wird abwechselnd eine Teilmenge der Kantenmenge ausgewählt und auf dieser ein maximales Matching bestimmt, das am Ende (d.h. wenn die Kantenteilmenge nicht mehr verändert wird) ein optimales Matching ist.

Matchingprobleme auf nicht bipartiten Graphen

Auch für nicht bipartite Graphen basiert die Suche nach maximalen, bzw. optimalen Matchings auf der Konstruktion vergrößernder Wege. Allerdings sind diese nicht mehr so leicht zu finden wie im bipartiten Fall. Der erste polynomiale Algorithmus zur Lösung des allgemeinen maximalen, bzw. optimalen Matchingproblems wurde 1965 von *Edmonds* angegeben.

4.3.3 Das Transportproblem

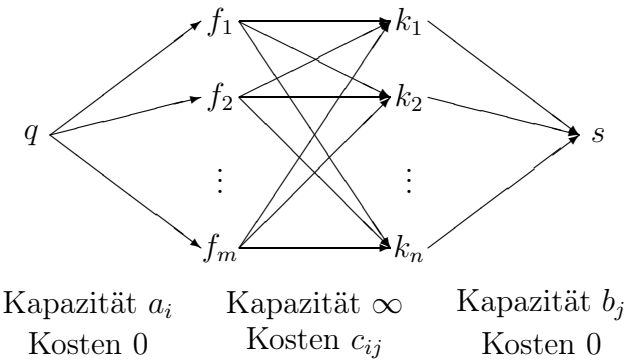
Unter dem Transportproblem versteht man folgende Aufgabenstellung:

Ein Gut wird in m Fabriken in den Mengen $a_1, \ldots, a_m$ hergestellt und von n Kunden in den Mengen $b_1, \ldots, b_n$ nachgefragt. Der Transport einer Men-

geneinheit dieses Gutes von der Fabrik i zum Kunden j verursacht Kosten in Höhe von c_{ij}. Bestimme die Anzahlen x_{ij} der zur vollständigen Befriedigung der Nachfrage von den Fabriken $i = 1, \ldots, m$ zu den Kunden $j = 1, \ldots, n$ zu transportierenden Mengeneinheiten so, dass die Summe der entstehenden Transportkosten minimal ist. (Falls dabei nicht $\sum_{i=1}^{m} a_i = \sum_{j=1}^{n} b_j$ gilt, fügt man noch einen virtuellen Anbieter, bzw. einen virtuellen Nachfrager hinzu, der mit Transportkosten Null die fehlenden Mengen liefert, bzw. die überschüssigen Mengen abnimmt.)

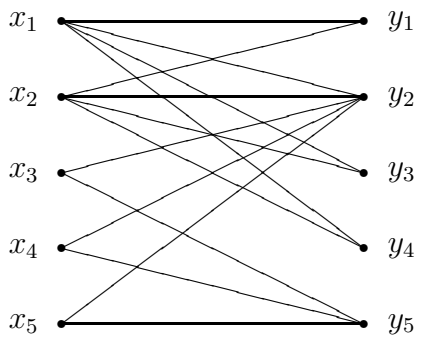
Kostenminimales Flußproblem

Die Quelle q ist durch gerichtete Kanten der Kapazität a_i mit den Fabriken f_i verbunden, ebenso die Konsumenten k_j mit der Senke s durch gerichtete Kanten der Kapazität b_j. Die Kosten eines Flusses längs dieser Kanten sind jeweils Null. Von jeder Fabrik verläuft je eine gerichtete Kante der Kapazität ∞ zu jedem Konsumenten k_j. Die Kosten pro Mengeneinheit des Flusses von f_i nach k_j betragen c_{ij}. Gesucht ist ein kostenminimaler Fluß von q nach s mit der Flußstärke $y = \sum_{i=1}^{m} a_i$.



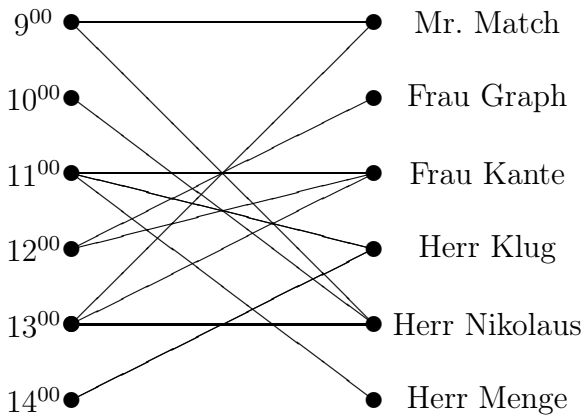
Aufgaben

1. Finden Sie ein maximales Matching in dem bipartiten Graphen



Bestimmen Sie dabei vor Anwendung der Ungarischen Methode zunächst ein Matching mit Hilfe des Greedy-Algorithmus. Untersuchen Sie außerdem mit Hilfe des Heiratssatzes, ob der Graph ein perfektes Matching besitzen kann.

2. **Vortragsprogramm** (? Punkte)
Auf einem Kongress über Graphentheorie soll ein Vortragsprogramm aufgestellt werden. Leider sind nicht alle Vortragenden zu jeder Zeit frei. Der folgende Graph zeigt ihre Verfügbarkeit. Lässt sich unter diesen Bedingungen ein Programm aufstellen ? Verwenden Sie dazu die ungarische Methode zusammen mit dem Greedy Algorithmus.



Kapitel 5

Tourenprobleme

Die in diesem Kapitel vorgestellten Aufgabenstellungen befassen sich mit Reisen einer Person auf einem als Wegenetz aufgefaßten **zusammenhängenden Graphen**. Dies gilt auch dann, wenn wir der einfachheit halber nur Graph schreiben! Es wird dabei unterschieden, ob bei diesen Reisen alle Kanten oder alle Ecken genau einmal passiert werden müssen.

5.1 Kantenbezogene Aufgaben

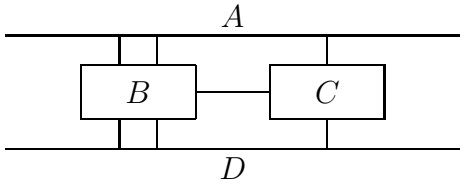
In diesem Abschnitt lassen wir ausdrücklich auch nicht schlichte zusammenhängende Graphen zu.

5.1.1 Eulertouren

Als Geburtsstunde der Graphentheorie gilt *Leonhard Eulers* Lösung des nachfolgenden Problems aus dem Jahre 1736.

Das Königsberger Brückenproblem

Die Stadt Königsberg liegt auf beiden Ufern des Pregels, in dessen Mitte außerdem im Stadtgebiet noch zwei Inseln liegen. Diese vier Gebiete sind durch sieben Brücken miteinander verbunden:



Ist es möglich, von einem der vier Gebietsteile A, B, C oder D ausgehend, jede der Brücken genau einmal zu überqueren und sich am Ende dieser Überquerungen wieder am Ausgangspunkt einzufinden?

Aufgrund dieser Aufgabenstellung sind die folgenden Begriffsbildungen mit dem Namen Euler verbunden:

Definition 5.1 (Eulertour und Eulerpfad)

- *Eine geschlossene Kantenfolge, die jede Kante eines Graphen genau einmal enthält, heißt eine* Eulertour.
- *Ein Graph, der eine Eulertour besitzt, heißt ein* eulerscher Graph.
- *Eine Kantenfolge, die jede Kante eines Graphen genau einmal enthält und nicht geschlossen ist, heißt ein* Eulerpfad.
- *Eine geschlossene Kantenfolge heißt ein* Eulerkreis*, wenn alle Kanten $e_1, \ldots, e_k$ voneinander verschieden sind und $v_0 = v_k$ gilt.*

Es gilt dann:

Satz 5.1

- *Ein ungerichteter Graph besitzt genau dann eine Eulertour, wenn jede Ecke einen geraden Grad besitzt.*
- *Ein ungerichteter Graph besitzt genau dann einen Eulerpfad, wenn genau zwei Ecken einen ungeraden Grad besitzen. Diese beiden Ecken sind die erste und die letzte Ecke des Eulerpfads.*

Beweis: Da bei einer Eulertour jede Ecke und bei einem Eulerpfad jede Ecke mit Ausnahme der ersten und letzten genauso oft erreicht wie verlassen werden muß, sind die angegebenen Bedingungen für die Existenz dieser Kantenfolgen notwendig. Dass sie dafür auch hinreichend sind, ergibt sich aus der weiter unten angegebenen Verfahrensvorschrift. q.e.d.

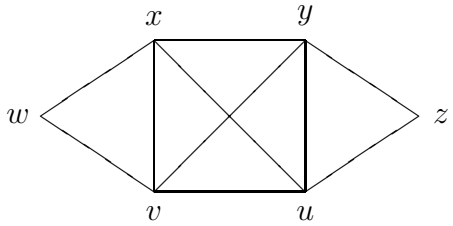
Aufgrund dieses Satzes ist klar, dass die Königsberger Brücken keinen Spaziergang der oben angegebenen Art zulassen. In dem Graphen, dessen Ecken und Kanten die vier Gebietsteile A, B, C und D und die sie verbindenden Brücken darstellen, haben alle vier Ecken einen ungeraden Grad. Es gibt also weder eine Eulertour noch einen Eulerpfad.

Die Existenz einer Eulertour oder eines Eulerpfads erlaubt es, einen Graphen „in einem Strich“ zu zeichnen. Bei einer Eulertour darf man dazu an einer beliebigen Ecke beginnen, bei einem Eulerpfad muß man dazu an einer der Ecken mit ungeradem Grad beginnen.

Praktische Ermittlung einer Eulertour

Die einfachste Methode zur expliziten Angabe einer Eulertour in einem Graphen ohne Ecken ungeraden Grades besteht darin, ausgehend von einer beliebigen Ecke „irgendwie“ den Graphen zu durchlaufen und jede passierte Kante zu notieren und dann aus dem Graphen zu entfernen, bis „es nicht mehr weitergeht“. Man ist dann notwendigerweise wieder bei der Ausgangsecke angekommen und hat also eine geschlossene Kantenfolge gefunden. Bei jeder Ecke dieser Kantenfolge wurde der Grad um eine gerade Zahl erniedrigt. Falls diese Kantenfolge noch nicht alle Kanten enthält, verfährt man mit jeder Komponente des verbliebenen Graphen genauso, bis schließlich eine Menge von geschlossenen Kantenfolgen entsteht, die jede Kante genau einmal enthält. Je zwei geschlossene Kantenfolgen, die eine Ecke gemeinsam haben, können zu einer einzigen geschlossenen Kantenfolge verschmolzen werden (beim Durchlaufen der einen Kantenfolge wird die andere „eingeschoben“, sobald erstmals eine ihrer Ecken erreicht wird), so dass schließlich aus diesen geschlossenen Kantenfolgen eine einzige geschlossene Kantenfolge, die jede Kante genau einmal enthält, also eine Eulertour, entsteht.

Beispiel 5.1 *Es sei folgender Graph gegeben:*



Beim Durchlaufen der Kanten kann man etwa die geschlossenen Kantenfolgen erhalten: w, x, v, w; x, y, v, u, x; y, u, z, y *die sich verschmelzen lassen zu* $w, x, y, u, z, y, v, u, x, v, w$

Dem Zerfallen der Eulertour in verschiedene Kantenfolgen kann man entgehen, wenn man bei der Bildung der ersten Kantenfolge so weit wie möglich vermeidet, eine Schnittkante des jeweils verbliebenen Graphen auszuwählen.

Graphen, die einen Eulerpfad besitzen, kann man mit dem vorstehenden Verfahren behandeln, wenn man zunächst die beiden Ecken ungeraden Grades durch eine zusätzliche Kanten verbindet, dann eine Eulertour konstruiert und schließlich die Zusatzkante hieraus wieder entfernt.

Zum Finden einer Eulertour existieren mehrere Verfahren. Der Algorithmus von Fleury, der im nächsten Abschnitt vorgestellt wird, stammt aus dem Jahr

1883 und verfolgt einen sehr einfachen Ansatz, weshalb er eine Laufzeit von der Größenordnung $\mathcal{O}(|E|^2)$ hat. Effizienter ist der Algorithmus von Hierholzer, der eine Eulertour in linearer Zeitkomplexität berechnet. Dieser basiert darauf, dass sich ein eulerscher Graph in paarweise kantendisjunkte Kreise zerlegen lässt.

Algorithmus von Fleury

Das im vorangegangenen Abschnitt beschriebene Verfahren basiert auf dem Satz, dass ein zusammenhängender Graph G dann und nur dann eulersch ist, wenn G die Vereinigung von kantendisjunkten Zyklen ist.

In diesem Abschnitt soll ein Verfahren vorgestellt werden, das direkt eine Eulertour angibt. Der wesentliche Unterschied zu dem Verfahren im vorangegangenen Abschnitt ist, dass bei der Wahlmöglichkeit von mehreren Kanten keine Schnittkanten gewählt werden.

Algorithmus 5.1 *Gegeben sei ein eulerscher Graph $G = (V, E)$. Ausgabe ist die Eulertour $W_{|E|}$.*

Schritt 1: *Man wähle eine beliebige Ecke v_0 in G und setze $W_0 = v_0$.*

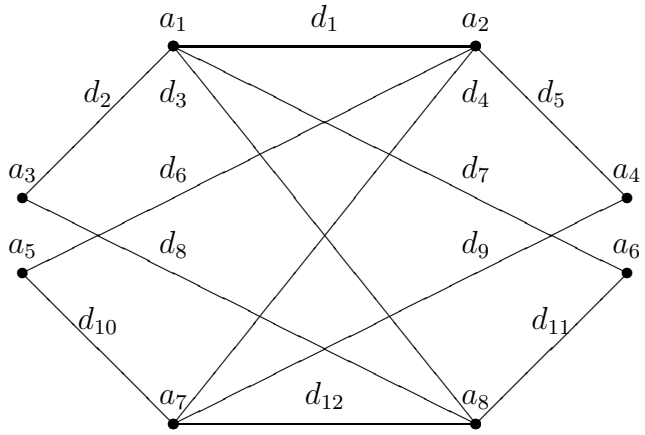
Schritt 2: *Wenn der Kantenzug $W_i = v_0 e_1 v_1 \dots e_i v_i$ gewählt worden ist (so dass alle $e_1 \dots e_i$ unterschiedlich sind), wähle man eine von $e_1 \dots e_i$ verschiedene Kante e_{i+1}, so dass*

1. *e_{i+1} inzident mit v_i ist und*
2. *ausgenommen, es gibt keine Alternative, e_{i+1} keine Schnittkante[1] des Teilgraphen $G \setminus \{e_1, \dots, e_i\}$ ist.*

Schritt 3: *Man beende den Algorithmus, wenn W_i jede Kante von G beinhaltet. Andernfalls ist Schritt 2 zu wiederholen.*

Beispiel 5.2 *Der Algorithmus wird für folgenden Graphen veranschaulicht. Die einzelnen Schritte möge der Leser durch Wegstreichen (Markieren, Ausradieren) der jeweils gewählten Kante selber nachvollziehen.*

[1] Ist nur noch eine Kante da, ist diese Schnittkante und sie muss gewählt werden. Sind mindestens zwei Kanten da, muss bei der Wahl einer dieser Kanten geprüft werden, ob diese eine Schnittkante ist[2]. Sollte dies der Fall sein, wird (bedenkenlos, denn es kann nur eine Schnittkante geben!) die andere Kante gewählt. Schnittkanten können naiv per (Tiefen-/Breiten-)Suche identifiziert werden: Gibt es einen anderen Weg von der Zielecke zur Startecke ?



Eine mögliche Tour wäre (es sind hier nur die Kanten aufgeführt):

$$W_{12} = d_1 d_5 d_9 d_4 d_6 d_{10} d_{12} d_3 d_7 d_{11} d_8 d_2.$$

Wir stellen nun noch eine Alternative vor, in der die Überprüfung auf Schnittkante wegfällt: Der Algorithmus von Hierholzer.

Algorithmus 5.2 *Gegeben sei ein eulerscher Graph $G = (V, E)$. Ausgabe ist die Eulertour $\mathcal{K}$.*

Schritt 1: *Man wähle einen beliebigen Knoten v_0 in G und konstruiere von v_0 ausgehend einen Unterkreis (Eulerkreis) K in G, der keine Kante in G zweimal durchläuft.*

Schritt 2: *Wenn $\mathcal{K}$ eine Eulertour ist, gehe zu 4. Andernfalls: gehe zu Schritt 3.*

Schritt 3:

1. *Vernachlässige*[3] *alle Kanten des Eulerkreises $\mathcal{K}$.*
2. *Eine beliebige Ecke von $\mathcal{K}$, zu der nicht vernachlässigte Kanten inzident sind*[4]*, startet man einen weiteren Eulerkreis $\mathcal{K}'$ (analog Schritt 1).*
3. *Füge in $\mathcal{K}$ den zweiten Kreis $\mathcal{K}'$ ein, indem die Startecke von $\mathcal{K}'$ in $\mathcal{K}$ durch alle Ecken von $\mathcal{K}'$ ersetzt wird*[5]*. Gehe nun zu dem Anfang von Schritt 2.*

Schritt 4: *Gebe $\mathcal{K}$ als Eulertour aus.*

[3] Z.B. durch eine Markierung als verbraucht bzw. besucht.

[4] Kanten, die nicht in $\mathcal{K}$ enthalten sind.

[5] Kann z.B. dadurch realisiert werden, dass man die Ecke explizit durch den Eulerkreis ersetzt.

Der Algorithmus von Fleury arbeitet wegen dem Test auf Schnittkante nicht nur lokal: man benötigt mehr Informationen vom Graphen, als die aktuelle Ecke und deren inzidenten Kanten, um diesen Test durchführen zu können. Hierholzer arbeitet dagegen nur lokal, da er an der aktuellen Ecke nur die inzidenten Kanten benötigt, um eine Wahl zu treffen. Lediglich bei den bisher erzeugten Ergebnissen ist er nicht lokal, da er eine Ecke auswählen muß, an der es weiter gehen kann. Jedoch kann diese auch während dem einfachen Durchlauf schon ermittelt werden. Dieser Unterschied kann bei sehr großen Graphen wichtig sein, da der Algorithmus von Hierholzer sich zunächst auf den im Arbeitspeicher liegenden Teil einschränken kann.

5.1.2 Das Chinesische Briefträgerproblem

Der Name dieses Problems geht auf den Chinesen *Mei-ko Kwan* zurück, der es 1962 formulierte:

Ein Postbote soll in seinem Zustellbezirk jede Straße (mindestens) einmal entlanggehen[6]. Insgesamt möchte er einen möglichst kurzen Weg zurücklegen (d.h. er möchte möglichst selten eine Straße zweimal durchlaufen müssen). Wie soll er seine Tour durch den Zustellbezirk planen?

Graphentheoretische Formulierung

Wir können den Zustellbezirk als Graphen darstellen, in dem die Kanten die Straßen und die Ecken Kreuzungen, Einmündungen und Endpunkte von Sackgassen darstellen. Jede Kante ist dann mit der Länge des Straßenabschnitts zwischen ihren Ecken bewertet. Der Postbote kann sicherlich diesen Graphen „in einem Zug“ durchlaufen, wenn wir jede Kante verdoppeln (d.h. durch zwei parallele Kanten ersetzen). Zu untersuchen ist mithin, ob bereits die Verdoppelung nur eines Teils der Kanten dem Graphen diese Eigenschaft verschaffen kann. Die Aufgabe lautet daher:

Finde in diesem Graphen $G(V, E)$ eine Kantenmenge mit minimaler Kantenbewertungssumme, durch deren Verdoppelung der Graph eulersch wird.

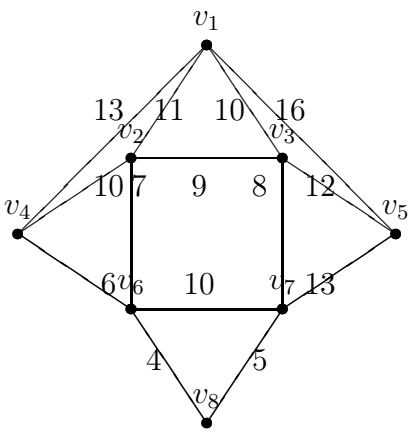
Spezialfall

Um die Idee der Verdopplung von Kanten zu verdeutlichen, wird in diesem Abschnitt eine Lösung vorgestellt, die den speziellen Fall betrachtet, dass nur

[6]Falls es in dem Zustellbezirk Straßen gibt, die nicht durchlaufen werden müssen, da an ihnen niemand wohnt, die aber durchlaufen werden dürfen, weil sie möglicherweise den Weg zwischen zwei bewohnten Gebieten verkürzen, wird die Aufgabenstellung als „Problem des Landbriefträgers“ bezeichnet. Dieses Problem ist schwieriger zu lösen als das vorliegende.

genau zwei Ecken einen ungeraden Grad haben. Im nachfolgenden Abschnitt wird eine allgemeineres Lösungsverfahren vorgestellt.

Wir betrachten folgenden Graphen:



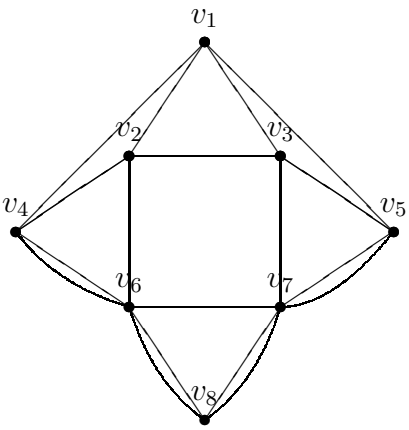
In diesem Graphen haben die beiden Ecken v_4 und v_5 einen ungeraden Grad. Mittels des Algorithmus von Dijkstra wird nun der kürzeste Weg von v_4 nach v_5 ermittelt[7].

v_4	v_1	v_2	v_3	v_6	v_7	v_8	v_5	T
0	∞	∞	∞	∞	∞	∞	∞	$\{v_4, v_1, v_2, v_3, v_6, v_7, v_8, v_5\}$
	13,v_4	10,v_4	∞	6,v_4	∞	∞	∞	$\{v_1, v_2, v_3, v_6, v_7, v_8, v_5\}$
	13,v_4	10,v_4	∞		16,v_6	10,v_6	∞	$\{v_1, v_2, v_3, v_7, v_8, v_5\}$
	13,v_4		19,v_2		16,v_6	10,v_6	∞	$\{v_1, v_3, v_7, v_8, v_5\}$
	13,v_4		19,v_2		15,v_8		∞	$\{v_1, v_3, v_7, v_5\}$
			19,v_2		15,v_8		29,v_1	$\{v_3, v_7, v_5\}$
			19,v_2				28,v_7	$\{v_3, v_5\}$
							28,v_7	$\{v_5\}$

Da $v_4v_6v_8v_7v_5$ der kürzeste Weg[8] von v_4 nach v_5 ist (wie aus der Tabelle hervorgeht), verdoppeln wir jede Kante dieses Weges und erhalten den nachfolgenden (bewerteten) Eulerschen Graphen. Die verdoppelten Kanten haben die gleiche Bewertung wie ihre „Doppelgänger“.

[7]Es ist zu beachten, dass die Anwendung des Algorithmus von Dijkstra den kürzesten Weg von v_4 zu allen anderen Ecken liefert!

[8]„kurz“ bezieht sich hier auf die Kantengewichte und nicht auf die Anzahl der Kanten!



Lösungsverfahren: Algorithmus von Edmonds

Ein polynomialer Algorithmus zur Lösung dieses Problems wurde von *Edmonds* angegeben.

Algorithmus 5.3 *Gegeben ein Graph G.*

1. *Finde die Menge U aller Ecken ungeraden Grades in G und bestimme für je zwei dieser Ecken u_i, u_j die Länge w_{ij} des kürzesten Weges zwischen ihnen.*

2. *Konstruiere den vollständigen Graphen $K_{|U|}$ mit Eckenmenge U und Kantenbewertungen $max - w_{ij}$, wobei max eine hinreichend große Zahl ist.*

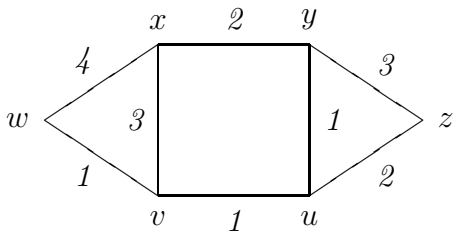
3. *Bestimme in diesem vollständigen Graphen ein Matching M mit maximaler Kantengewichtssumme.*

4. *Für jede Kante $u_iu_j \in M$ verdoppele in G die Kanten eines kürzesten Weges von u_i nach u_j.*

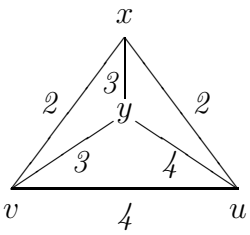
5. *Der so geänderte Graph G ist jetzt eulersch und kann von einer beliebigen Ecke ausgehend „in einem Zug“ durchlaufen werden.*

Dieser Algorithmus ist zwar mit den Kenntnissen aus diesem Text nicht allgemein durchführbar, da kein Verfahren zur Bestimmung von Matchings in nichtbipartiten Graphen behandelt wurde. Er wurde aber trotzdem an dieser Stelle angegeben, da er zeigt, wie Lösungsverfahren aus früheren Kapiteln miteinander kombiniert werden können, um neue Aufgabenstellungen zu lösen.

Beispiel 5.3 *Dieser Graph besitzt vier Ecken mit ungeradem Grad:*

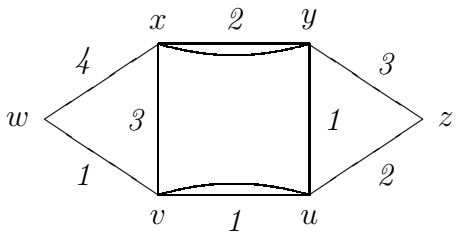


Der Graph K_4 mit den Kantenbewertungen $5 - w_{ij}$ lautet dann:



Da er genau drei Matchings besitzt, kann man diese noch „von Hand“ inspizieren (`xy uv`*: 3+4=7;* `uy vx`*: 4+2=6;* `vy ux`*: 3+2=5) und erhält als optimale Wahl die Paarungen uv und xy.*

Damit wird der vorgelegte Graph durch folgende Erweiterung mit minimalem Zusatzaufwand eulersch:

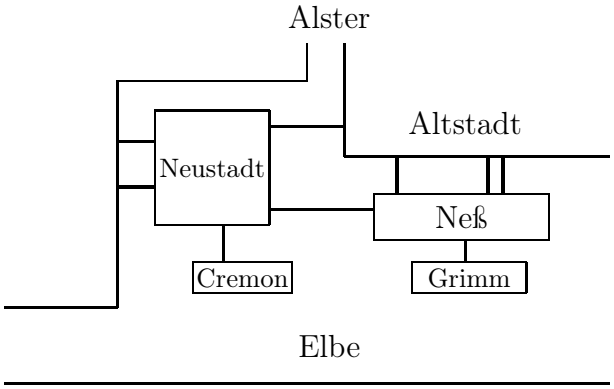


Gerichtete Kanten

Da in der Aufgabenstellung von einem Postboten die Rede war und Postboten ihren Dienst häufig zu Fuß ausüben, haben wir stillschweigend unterstellt, dass wir es mit einem ungerichteten Graphen zu tun haben. In der Tat kommen jedoch häufig auch gerichtete Kanten ins Spiel, etwa wenn nicht die Postzustellung, sondern die Müllabfuhr geplant werden soll und sich die Müllwagen an Einbahnstraßenregelungen halten müssen. Falls in diesen Situationen das gesamte Straßennetz als gerichteter Graph dargestellt werden kann, ist ebenfalls ein polynomialer Algorithmus zur Lösung des Postbotenproblems bekannt. Falls jedoch die Darstellung des Straßennetzes als gemischter Graph erforderlich ist, gibt es (noch) keinen polynomialen Algorithmus.

Aufgaben

1. Zu Beginn des 13. Jahrhunderts, als Hamburg noch etwas übersichtlicher war als heute, erstreckte sich das Stadtgebiet über Teile des Alsterufers und mehrere Inseln in der Alstermündung. Die verschiedenen Gebietsteile waren durch insgesamt 9 Brücken untereinander verbunden:



   Wäre es damals möglich gewesen, auf einem Sonntagsspaziergang alle Brücken genau einmal zu passieren? Wie ändert sich die Lage, wenn man zwischen Cremon und Grimm auf einem Boot übersetzen kann?

2. Formulieren Sie eine Bedingung für die Existenz einer Eulertour in einem gerichteten Graphen.

3. Finden Sie eine Eulertour in dem Graphen K_7.

4. **Kantenmodell** (? Punkte)

   (a) Kann man einen Draht so biegen, dass ein Kantenmodell eines Würfels entsteht? Wie sieht es bei einem Oktaeder (3-D Objekt mit 6 Ecken) aus ?

   (b) Sei $G = (V, E)$ ein eulerscher Graph mit $|E| = q$ Kanten. Weiter sei $S := \{v \in V \| \frac{1}{2} \cdot d(v)$ ist ungerade$\}$. Zeigen Sie: Ist q ungerade, dann hat S eine ungerade Anzahl von Elementen.

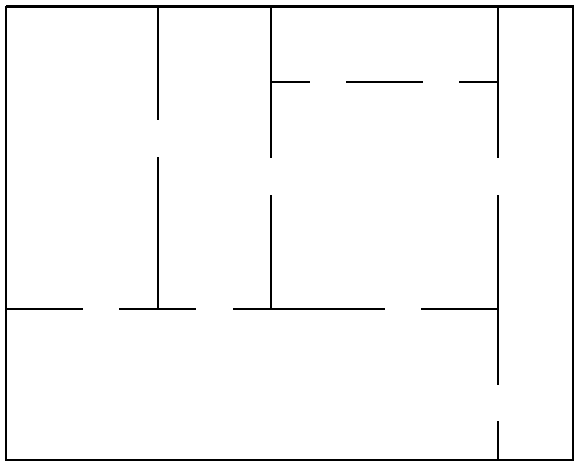
5. In dieser Aufgabe betrachten wir eine Verallgemeinerung der Domino-Aufgabe: Auf den Spielsteinen stehen die Zahlen 0 bis k ($k \in \mathbb{N}$) und es gibt für jedes Zahlenpaar einen Spielstein $i/j (i \leq j)$, also $0/0, 0/1, \ldots 0/k$, $1/1, 1/2 \ldots 1/k, 2/2, \ldots k/k$.

(a) Begründen Sie, dass die Aufgabe, sämtliche Steine in einer geschlossenen Kette unterzubringen, nur für gerade Zahlen k lösbar ist.

(b) Wenn k ungerade ist, werden die Steine $0/1, 2/3, 4/5, \ldots, (k-1)/k$ aus dem Dominospiel entfernt. Begründen Sie, dass die Domino-Aufgabe dennoch lösbar ist.

6. **Eulerpfad** (? Punkte)
Begründen Sie: Hat ein zusammenhängender Graph vier Ecken mit ungeradem Grad und sonst lauter Ecken mit geradem Grad, so kann man ihn in zwei Linien zeichnen, jede von ihnen ohne abzusetzen, und zwar so, dass insgesamt keine Strecke doppelt gezeichnet wird, d.h. man kann mit zwei Eulerpfaden den ganzen Graphen abdecken.

7. **Eulertour** (? Punkte)
Gegeben sei nachfolgender Grundriss einer Wohnung:



Ist es möglich in dieser Wohnung einen Rundgang zu machen, so dass man durch jede Tür genau einmal hindurch kommt ?

8. **Eulertour** (? Punkte)
Wenden Sie den Algorithmus von Fleury auf den Graphen in Abbildung 5.1 (Seite 130) an (bis zum Abbruch!). Fertigen Sie eine Dokumentation an, aus der der Ablauf des Algorithmus deutlich wird (Auswahl der Kanten).

9. **Eulertour** (? Punkte)
Seien $m, n \in \mathbb{N}$.

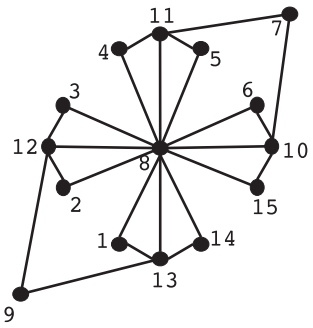


Abbildung 5.1: Euelertour gesucht!

(a) Für welche m, n hat der vollständige bipartite Graph $K_{m,n}$ eine Eulertour ?

(b) Sei $K_{n,n} \ominus M_n$ der Graph, der aus dem vollständigen bipartiten Graph $K_{n,n}$ durch Löschen (der Kanten) eines perfekten Matchings entsteht. Für welche n hat $K_{n,n} \ominus M_n$ eine Eulertour?

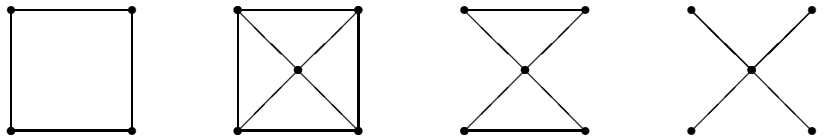
(c) Zeichnen Sie vier zusammenhängende Graphen mit jeweils 7 Ecken, von denen einer eine Eulertour und einen Hamiltonkreis besitzt, einer keine Eulertour und einen Hamiltonkreis besitzt, einer eine Eulertour und keinen Hamiltonkreis besitzt und einer keine Eulertour und keinen Hamiltonkreis besitzt. Geben Sie jeweils die Touren/Kreise an, sofern sie vorhanden sind, bzw. machen Sie deutlich, warum es sie nicht gibt.

5.2 Eckenbezogene Aufgaben

5.2.1 Hamiltonsche Kreise

Unter einem Hamiltonkreis (auch als Rundreise bezeichnet) versteht man einen Kreis, der jede Ecke eines vorgelegten Graphen genau einmal durchläuft[9]. Ein Hamiltonweg ist ein Weg, der jede Ecke eines vorgelegten Graphen genau einmal enthält.

[9]Die Aufgabe, in einem bestimmten Graphen einen solchen Kreis zu finden, wurde von *William R. Hamilton* im Jahre 1858 als Spiel veröffentlicht



Ein hamiltonscher Graph und ein eulerscher Graph sind unabhängig von einander. Die vorangestellten Beispiele zeigen dies: von links nach rechts sind die jeweiligen Graphen eulersch und hamiltonsch, nicht eulersch und hamiltonsch, eulersch und nicht hamiltonsch, sowie nicht eulersch und nicht hamiltonsch.

Für die Existenz eines Hamiltonkreises oder Hamiltonweges sind keine ähnlich universell anwendbare Bedingungen wie im Falle einer Eulertour oder eines Eulerpfades bekannt. Auch gibt es (noch) keinen polynomialen Algorithmus zur Konstruktion eines Hamiltonkreises oder Hamiltonweges.

Definition 5.2 *Ein* Hamiltonscher Weg *in einem Graphen G ist ein Weg, der jede Ecke von G enthält.*

Ein Hamiltonscher Kreis *(oder Hamiltonischer Zyklus) in einem Graphen G ist ein Kreis, der jede Ecke von G enthält.*

Ein Graph heißt hamiltonsch*, wenn er einen hamiltonschen Kreis enthält.*

Die Frage nach der Existenz eines Hamiltonkreises (oder kurz: das Hamiltonkreisproblem) ist ein Beispiel für eine Aufgabenstellung, deren Lösung für beliebige konkrete Daten stets nur einen von zwei Werten („ja“, bzw. „nein“) annehmen kann. Probleme dieser Art werden als Entscheidungsprobleme bezeichnet. Weitere Beispiele für Entscheidungsprobleme sind etwa die Fragen „Ist die Zahl n eine Primzahl?“ oder „Gibt es eine Belegung der Variablen x, y und z mit Wahrheitswerten, so dass ein bestimmter vorgelegter Boolescher Ausdruck $f(x, y, z)$ den Wert ‚wahr‘ annimmt“?

Im folgenden seien einige Sätze aufgeführt, die für spezielle Fälle das Entscheidungsproblem gelöst haben.

Satz 5.2 (Dirac) *Wenn G ein schlichter Graph mit n Ecken, mit $3 \leq n \in \mathbb{N}$, ist und der Grad jeder Ecke v von G $d(v) \geq \frac{n}{2}$ beträgt, dann ist G hamiltonsch.*

Betrachten wir die vollständigen Graphen K_n so gilt für jede Ecke, dass $d(v) = n - 1 \geq \frac{n}{2}$, d.h. alle vollständigen Graphen sind hamiltonsch und es gibt $\frac{(n-1)!}{2}$ unterschiedliche hamiltonsche Kreise!

Der Satz von Dirac beschreibt ein positives Kriterium. Der nachfolgende Satz ist zwar positiv formuliert, wird aber als Ausschlußkriterium verwendet.

Satz 5.3 *Löscht*[10] *man in einem hamiltonschen Graphen n Ecken, so zerfällt der Graph in höchstens n Teilgraphen.*

Kann man z.B. aus einem Graphen eine Ecke so löschen, das der Graph in zwei Teilgraphen zerfällt, so ist dieser Graph nicht hamiltonsch (Ausschlußkriterium)! Zerfällt ein Graph jedoch in maximal n Teilgraphen nach dem Löschen von n (beliebigen) Ecken, so muß der Graph nicht hamiltonsch sein, da in dem vorangegangenen Satz eine Implikation formuliert ist. Beispielsweise kann man in dem sogenannten Petersen-Graph beliebige n Ecken löschen, ohne dass der Graph in mehr als n Teilgraphen zerfällt, einen Hamilton-Kreis gibt es aber dennoch nicht.

Für den nachfolgenden wichtigen Satz muß der Begriff der Hülle eingeführt werden.

Definition 5.3 *Gegeben sei ein schlichter Graph G_0 mit n Ecken. Solange es zwei nicht adjazente Ecken v_1, v_2 in G_i gibt, so dass $d(v_1) + d(v_2) \geq n$ in G_i ist, verbinde man diese beiden Ecken, um einen neuen Obergraphen G_{i+1} zu bilden. Der letzte Obergraph, der so erhalten wird, heißt* Hülle *oder auch Hamiltonabschluss von G_0 und wird mit $c(G_0)$ bezeichnet.*

Die Bedeutung einer Hülle $c(G)$ ist durch das folgende Ergebnis gegeben.

Satz 5.4 (Bondy und Chvatal) *Ein schlichter Graph G ist dann und nur dann hamiltonsch, wenn seine Hülle $c(G)$ hamiltonsch ist.*

Die Hoffnung besteht also darin, mit der Hülle einen Graphen erzeugt zu haben, in der die Klärung, ob dieser hamiltonsch ist, einfach ist: z.B. wenn ein vollständiger Graph entsteht. Sollte der Graph aber so geartet sein, dass z.B. dieser die Hülle von sich selbst ist, hilft einem dieser Satz wenig.

Einen allgemeinen, naiven Algorithmus zum finden eines hamiltonschen Kreises kann man schon angeben. Daraus läßt sich allerdings nicht folgern, dass es sich hier um ein einfaches Problem handelt.

Algorithmus 5.4 (naiver Algorithmus) *Vorausgesetzt es gibt einen hamiltonschen Kreis, können wir ihn wie folgt finden:*

- *Wenn wir eine Ecke in den möglichen Kreis aufgenommen haben, löschen wir bei dem Vorgänger (nicht für die Startecke) alle nicht benutzten und mit dem Vorgänger inzidenten Kanten (, da wir diese Ecke nicht mehr besuchen werden).*

[10] Entfernen der Ecke inklusive der inzidenten Kanten.

Wenn eine der nachfolgenden Situationen eintritt (außer am Ende der Suche, wenn es sich nicht mehr vermeiden läßt), müßen wir wieder zum Vorgänger zurück und eine andere Fortsetzung wählen (Backtracking):

- *Es dürfen keine Ecken mit Grad 1 entstehen.*
- *Es darf kein nicht zusammenhängender Graph entstehen.*

Das Hamiltonkreisproblem ist ein Spezialfall des im folgenden Abschnitt vorgestellten Problems.

5.2.2 Das Rundreiseproblem

Die Aufgabenstellung dieses Problems wird üblicherweise in folgender „Einkleidung“ präsentiert:

Aufgabenstellung

Ein Handlungsreisender[11] im Ort v_1 will Kunden in den Orten $v_2, \ldots, v_n$ besuchen und dann nach Hause zurückkehren. Die Entfernung zwischen den Orten v_i und v_j sei jeweils $w_{ij} \geq 0$. In welcher Reihenfolge soll er die Orte besuchen, damit die insgesamt zurückgelegte Entfernung minimal wird?

Graphentheoretisch formuliert bedeutet dies: In einem vollständigen Graphen K_n mit Kantenbewertungen w_{ij} ist ein Hamiltonkreis mit minimaler Kantenbewertungssumme gesucht. Falls stets $w_{ij} = w_{ji}$ gilt (symmetrisches TSP), ist der betrachtete Graph ungerichtet, andernfalls (asymmetrisches TSP) wird jedes Eckenpaar durch zwei entgegengesetzt gerichtete Kanten verbunden.

Der Kundenkreis eines Handlungsreisenden ist für eine sinnvolle mathematische Behandlung in der Regel zu klein. Anders sieht es aus, wenn beispielsweise ein Punktschweißroboter auf möglichst kurzem Wege über die Oberfläche eines Werkstücks mit einer großen Anzahl von Schweißpunkten geführt werden soll oder wenn eine Platine in möglichst kurzer Zeit mit Bauteilen an vorgegebenen Positionen automatisch bestückt werden soll.

(Nicht)existenz eines polynomialen Algorithmus

Trotz intensiver jahrzehntelanger Forschung ist bis heute kein polynomialer Algorithmus für das TSP bekannt. Stattdessen wurde bewiesen, dass aus der Existenz eines polynomialen Algorithmus für das TSP die Existenz polynomialer Algorithmen für viele andere Probleme, für die trotz großer Anstrengungen

[11] In Anlehnung an den englischen Namen („Travelling Salesman Problem“) wird das Rundreiseproblem auch mit dem Kürzel TSP bezeichnet.

bisher noch keine solchen Algorithmen gefunden werden konnten, folgen würde (darunter das ILP[12], das Problem des längsten Weges, das Chinesische Postbotenproblem auf gemischten Graphen und das Hamiltonkreisproblem). Es ist deshalb eine weitverbreitete Vermutung, anzunehmen, dass es keinen polynomialen Algorithmus für das TSP gibt. Aus diesem Grunde ist viel Arbeit in die Verbesserung exponentieller Algorithmen investiert worden, mit denen inzwischen Probleme mit bis zu gut tausend Ecken in vertretbarer Zeit gelöst werden können[13].

P und NP

Die Menge aller Entscheidungsprobleme, für die ein polynomialer Algorithmus existiert (z.B. die Frage nach der Existenz eines Eulerpfades), wird als Klasse P bezeichnet. Die Menge aller Entscheidungsprobleme, die polynomial lösbar sind, falls das Hamiltonkreisproblem polynomial lösbar ist, wird als Klasse NP bezeichnet. (Insbesondere gilt $P \subseteq NP$.) Diejenigen Elemente von NP, aus deren polynomialer Lösbarkeit auch umgekehrt die polynomiale Lösbarkeit des Hamiltonkreisproblems folgen würde, heißen NP-vollständige Probleme. Hierzu gehören beispielsweise die Frage nach der Erfüllbarkeit eines booleschen Ausdrucks, oder danach, ob in einem vorgelegten vollständigen gerichteten Graphen eine bestimmte Rundreisenlänge unterboten werden kann, oder danach ob ein vorgelegtes ILP eine zulässige Lösung besitzt. Optimierungsprobleme, aus deren polynomialer Lösbarkeit die polynomiale Lösbarkeit des Hamiltonkreisproblems oder eines anderen NP-vollständigen Problems folgen würde, heißen NP-schwer. Hierzu gehören alle oben als (noch) nicht polynomial lösbar genannten Probleme.

Wie nun vorgehen, wenn ein Problem als NP-schierig klassifiziert ist ? Meist ist ist in einer konkreten Anwendung gar nicht das allgemeine Problem zu lösen, sondern lediglich ein (polynomialer) Spezialfall. Das kann daran liegen, dass nur ein eingeschränktes Vokabular der zugrundeliegenden formalen (Repräsentations-)Sprache benutzt wird. Z.B. haben konkrete Graphen aus der Anwendung keine Schlingen, obwohl die Sprache diese i.allg. zuläßt. Das kann aber auch daran liegen, dass Problemparamter identifiziert werden können, die innerhalb der Anwendung als konstant oder logarithmisch beschränkt angenommen werden können und damit zu einem polynomialen Zeitverhalten führen. So können konkrete Graphen aus der Anwendung z.B. immer genau 5

[12]Integer Linear Programming Problem: $a * xRb$ mit $R \in \{\leq, \geq, =\}$ und alle Zahlen sind aus $\mathbb{Z}$.

[13]Näheres siehe *E. L. Lawler et al.:* The Traveling Salesman Problem, John Wiley, Chichester - New York, 1985

Ecken besitzen oder jeder Grad der Ecken ist kleiner drei etc.

Im Falle von anwendungsunabhängigen Systemen, etwa ein allgemeiner TSP-Löser, kann man jedoch nicht davon ausgehen, dass die Anwendung schon so aussehen wird, dass „alles gut geht“. Hier sollte man zumindest in der Lage sein, die Bedingungen zu spezifizieren, unter denen das System zuverlässig arbeitet. Selbst wenn es also nicht möglich ist, anwendungsrelevante Spezialfälle zu identifizieren, ist die Analyse polynomialer Spezialfälle oft interessant, da sie uns Aufschluß darüber gibt, welche Aspekte eines Problems zu der „kombinatorischen Explosion“ führen. Diese „Grenzen“ sind dann die Grenzen für das angebotene System!

Das symmetrische TSP mit Dreiecksungleichung

Für den Fall, dass die Kantenbewertungen die Dreiecksungleichung erfüllen, d.h. wenn für beliebige drei unterschiedliche Zahlen $i, j, k = 1, \ldots, n$ gilt

$$w_{ij} \leq w_{ik} + w_{kj}$$

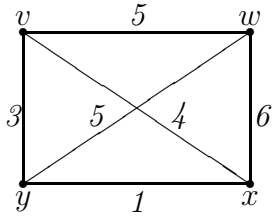
läßt sich das TSP immerhin näherungsweise mit einem polynomialen Algorithmus lösen. Der einfachste Algorithmus dieser Art bestimmt eine Rundreise, die höchstens das Doppelte der minimalen Länge aufweist:

1. Finde ein Minimalgerüst T in dem vollständigen Graphen K_n. (Die Summe der Kantenbewertungen des Minimalgerüsts ist höchstens so groß wie die Kantenbewertungssumme einer kürzesten Rundreise — vgl. Aufgabe.)

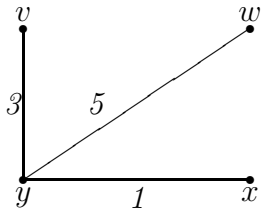
2. Verdoppele alle Kanten aus T. Dadurch wird T eulersch. Bestimme eine Eulertour in T und notiere die zugehörige Folge der Ecken. (Die Kantenbewertungssumme der Eulertour ist genau doppelt so groß wie die des Minimalgerüsts.)

3. Streiche in dieser Folge alle Ecken mit Ausnahme ihres ersten Auftretens und mit Ausnahme der letzten Ecke. Die verbleibende Eckenfolge definiert eine Rundreise im Graphen K_n. (Aufgrund der Dreiecksungleichung ist ihre Kantenbewertungssumme nicht größer als die der Eulertour.)

Beispiel 5.4 *Vorgelegt sei der Graph*



Ein Minimalgerüst ist



Eine Tour, die jede der Kanten des Minimalgerüsts genau zweimal durchläuft, ist beispielsweise

$$v, y, w, y, x, y, v$$

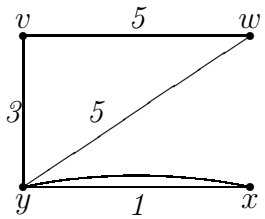
Durch Streichen von Ecken erhalten wir die Rundreise

$$v, y, w, x, v$$

mit der Gesamtlänge 18.

Dieser Algorithmus kann dahingehend verbessert werden, dass er eine Rundreise liefert, die garantiert höchstens 1.5-mal so lang ist wie der Minimalwert. Dazu werden die Kanten von T nicht einfach verdoppelt, sondern es wird zwischen den Ecken, die in T einen ungeraden Grad haben, in dem entsprechenden Untergraphen von K_n ein perfektes Matching mit minimaler Kantenbewertungssumme bestimmt, dessen Kanten dann zu T hinzugefügt werden.

Beispiel 5.5 *(Fortsetzung) In dem obigen Minimalgerüst haben alle Ecken einen ungeraden Grad. Deshalb muß das Matching im Ausgangsgraphen K_4 gesucht werden. Inspektion „von Hand“ liefert die Kanten vw und xy. Eine Eulertour in dem so ergänzten Minimalgerüst*



ist beispielsweise

$$y, x, y, w, v, y$$

Durch Streichen von Ecken erhalten wir die Rundreise

$$y, x, w, v, y$$

mit der Gesamtlänge 15.

Leider ist das Problem, für *allgemeine* Kantenbewertungen w_{ij} einen Algorithmus anzugeben, der das TSP mit einem vorgegebenen relativen Fehler löst, NP-schwer, da mit einem solchen Algorithmus das Hamiltonkreisproblem gelöst werden kann.

Methode der Einführung der dichtesten Ecke

Eine andere Methode, die die gleiche Qualität bzgl. Laufzeit ($O(n^2)$) und Resultat (besser als das doppelte vom optimalen Minimum) hat und ebenfalls auf der Dreiecksungleichung beruht, ist die Methode der Einführung der dichtesten Ecke[14].

Für den Algorithmus der Einführung der dichtesten Ecke spielt der Abstand einer Ecke v von einer Kantenfolge W eine entscheidende Rolle. Dieser ist definiert als

$$d(v, W) = min\{d(v, u) \| u \text{ ist eine Ecke von } W\}$$

$d(v, u)$ bezeichnet dabei die Kosten der zugehörigen Kante. Die Ecke v, die nicht in W liegt, wird als *dichteste* Ecke zu W bezeichnet, wenn für jede andere nicht in W befindliche Ecke x $d(v, W) \leq d(x, W)$ gilt.

Algorithmus 5.5 *Gegeben sei ein Graph $K_n = (V, E)$ mit $V = \{v_1, v_2, \ldots v_n\}$. Der aktuelle gewählte Kreis W_i wird stets neu nummeriert und ist gegeben durch $W_i = u_1 u_2 \ldots u_i u_1, u_j \in V$. Diese neue Nummerierung verändert nicht die Reihenfolge der bisher gewählten Ecken!*

Schritt 1: *Man wähle eine beliebige Ecke $u_1 = v_j \in V$ als Startecke und setze $W_1 = u_1$.*

[14]Der *nearest insertion algorithm* stimmt mit diesem hier nicht überein: dort wird immer nur von der zuletzt behandelten Ecke der dichteste Nachbar gesucht. Hier wird von allen Ecken in der aktuellen Tour der dichteste Nachbar gesucht!

Schritt 2: *Aus der Zahl der $n-i$ Ecken, die bisher noch nicht gewählt worden sind, ermittle man eine Ecke $u_{i+1} = v_k$, die am dichtesten zu W_i liegt. Sei $W_i = u_1u_2 \dots u_iu_1$. Man bestimme dann, welche der Kantenfolgen (Kreise) $u_1u_{i+1}u_2u_3 \dots u_iu_1$, $u_1u_2u_{i+1}u_3 \dots u_iu_1$, $\dots$ $u_1u_2u_3 \dots u_iu_{i+1}u_1$ die kürzeste ist. Es sei W_{i+1} die kürzeste Kantenfolge. Man nummeriere sie, wenn nötig, neu als $u_1 \dots u_{i+1}u_1$. Man setze $i := i + 1$.*

Schritt 3: *Wenn W_i alle Ecken beinhaltet, beende den Algorithmus, sonst führe Schritt 2 aus.*

Den Algorithmus der Einfügung des dichtesten Knotens für das Problem des Handlungsreisenden wird an dem vollständigen bewerteten Graphen in Abbildung 5.2 (Seite 138) (bis zum Abbruch!) aufgezeigt. Es wird jeweils der

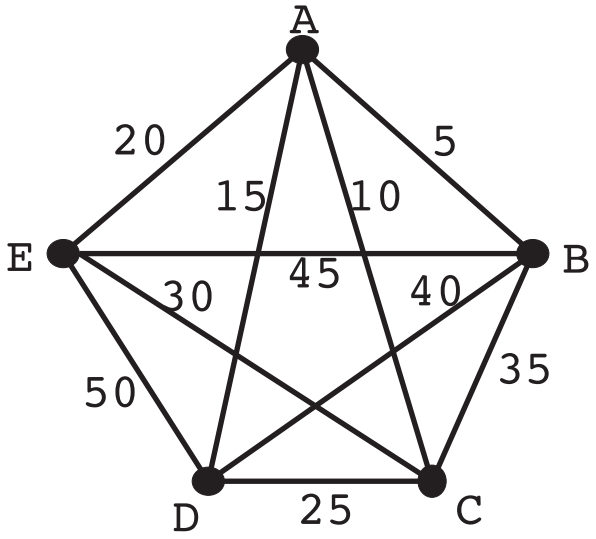


Abbildung 5.2: Problem des Handlungsreisenden

dichteste Knoten gewählt.

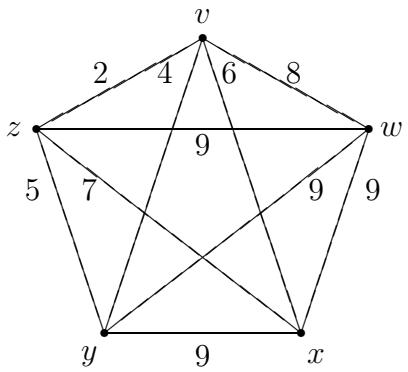
Schritt	Ablauf
1	$u_1 := A$
2	$d(B,W) = 5, d(C,W) = 10, d(D,W) = 15, d(E,W) = 20$:
	$u_2 := B$, so dass $W_2 = ABA = 10$
3	$d(C,W) = 10, d(D,W) = 15, d(E,W) = 20$:
	$u_3 := C$, Folgende Zyklen werden betrachtet:
	$u_1u_2u_3u_1 = ABCA = 50$
	$u_1u_3u_2u_1 = ACBA = 50$
	Wähle $W_3 = ABCA = 50$
4	$d(D,W) = 15, d(E,W) = 20$:
	$u_4 := D$. Folgende Zyklen werden betrachtet:
	$u_1u_2u_3u_4u_1 = ABCDA = 80$
	$u_1u_2u_4u_3u_1 = ABDCA = 80$
	$u_1u_4u_2u_3u_1 = ADBCA = 100$
	Wähle $W_4 = ABCDA := u_1u_2u_3u_4u_1$
5	$d(E,W) = 20$:
	$u_5 := E$. Folgende Zyklen werden betrachtet:
	$u_1u_2u_3u_4u_5u_1 = ABCDEA = 135$
	$u_1u_2u_3u_5u_4u_1 = ABCEDA = 135$
	$u_1u_2u_5u_3u_4u_1 = ABECDA = 120$
	$u_1u_5u_2u_3u_4u_1 = AEBCDA = 140$
	Wähle $W_5 = ABECDA := u_1u_2u_3u_4u_5u_1$

Damit ist der ziemlich kurze Kreis $W_5 = ABECDA$ mit den Kosten von 120 gefunden. Wenn man alle möglichen Permutationen untersucht, ergibt sich folgendes Spektrum: 2-mal Kosten 120, 4-mal Kosten 135, 4-mal Kosten 140 und 2-mal Kosten 155 (z.B. $W_{max} = ACBDEA$. Obzwar der Algorithmus nicht optimal arbeitet, hat er das Optimum gefunden!

Aufgaben

1. Zeigen Sie, dass das Hamiltonkreisproblem in einem vorgegebenen ungerichteten Graphen $G(V,E)$ ein Spezialfall des TSP ist, indem Sie G durch Einfügen weiterer Kanten in den vollständigen Graphen $K_{|V|}$ überführen, für alle Kanten $v_iv_j \in E$ setzen $w_{ij} := 1$ und für alle übrigen Kanten $w_{ij} := 2$ und dann für den so bewerteten vollständigen Graphen das TSP lösen.

2. Überzeugen Sie sich davon, dass die Kantenbewertungen in dem Graphen K_5 die Dreiecksungleichung erfüllen, und geben Sie eine Rundreise an, deren Länge höchstens das Doppelte der minimalen Länge beträgt:



3. Warum ist die Länge jeder Rundreise mindestens so groß wie die Kantengewichtssumme eines Minimalgerüsts?

4. In wieweit müssen die Überlegungen dieses Abschnitts beim Auftreten negativer Kantenbewertungen abgewandelt werden?

5. Wieviele Rundreisen besitzt ein vollständiger gerichteter Graph mit n Ecken?

6. Zeichnen Sie

   (a) einen Digraphen, der nicht eulersch ist, obwohl der zugrunde liegende Graph eulersch ist.

   (b) einen Digraphen, der nicht Hamiltonsch ist, obwohl der zugrunde liegende Graph Hamiltonsch ist.

   (c) einen stark zusammenhängenden Digraphen, der nicht eulersch ist.

   (d) einen stark zusammenhängenden Digraphen, der nicht Hamiltonsch ist.

   Begründen Sie jeweils, warum die Forderung erfüllt wird.

7. Zeichnen Sie alle (bis auf Isomorphie gleiche) einfachen ungerichteten hamiltonschen Graphen

   (a) mit drei Ecken.

   (b) mit vier Ecken.

   (c) Warum ist der Zusatz „einfach“ in dieser Aufgabe wichtig ?.

   Begründen Sie, warum es jeweils keine anderen Graphen gibt, die die geforderte Bedingung erfüllen!

8. **hamiltonscher Graph** (? Punkte)
   Zeichnen Sie einen schlichten Graphen, in dem alle Ecken mindestens den Grad vier haben, und der nicht hamiltonsch ist. Begründen Sie Ihre Lösung!

9. Diplomaten aus 6 Ländern sollen zu einer Konferenz an einem runden Tisch Platz nehmen. Das ist ziemlich schwierig, da es Konflikte unter ihnen gibt, und zwar zwischen den Ländern A und B, zwischen B und C, zwischen B und E, zwischen C und D sowie zwischen E und F, und deshalb sollen ihre Vertreter nicht direkt nebeneinander sitzen. Kann der Protokollchef die Diplomaten so platzieren, dass diese Bedingung erfüllt ist ?

   Lösen Sie die Aufgabe, indem Sie das Problem in einem Graphen darstellen und auf das Problem zurück führen, dort einen Hamiltonschen Kreis zu finden. Geben Sie dann alle möglichen Hamiltonschen Kreise an.

10. **Sitzordnungen** (? Punkte)
    Sechs Personen sollen an einem runden Tisch Platz nehmen, so dass jeder zwischen zwei Personen sitzt, die er schon kennt. Die Personen heißen A, B, C, D, E und F. In den folgenden Gruppen kennt jeder jeden:

    $$A, B, D, E \qquad B, C, F \qquad A, F \qquad C, D$$

    Kann man die Personen wie gewünscht platzieren ? Geben Sie ggf. fünf mögliche Sitzordnungen an.

    Lösen Sie die Aufgabe, indem Sie das Problem in einem Graphen darstellen und auf das Problem zurück führen, dort einen Hamiltonkreis zu finden.

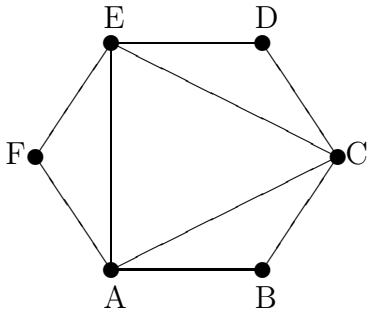
11. **Mäuse** (?? Punkte)
    Ein aus kleinen Käsewürfel der Maße $1 \times 1 \times 1$ zusammengesetzter Käsehaufen der Form eines Würfels mit den Maßen $3 \times 3 \times 3$ soll von einer Maus links unten beginnend Käsewürfel für Käsewürfel (also ohne „Sprünge“) aufgefressen werden (wobei der Mäuseengel dafür sorgt, dass der Käsehaufen dabei nicht zusammenfällt!)

    - Kann die Maus sich durch alle kleinen Käsewürfel mit jeweils Einerschritten (also den jeweiligen Käsewüfel komplett auffressend und nur zu einem horiziontalen/vertikalen benachbartem Käsewürfel weiter gehend) hindurch fressen und in der Mitte aufhören ? Geben Sie ggf. einen Weg an.

- Kann sie es schaffen wenn sie an einer anderen Stelle anfängt zu fressen ? Geben Sie ggf. einen Weg an.

Tipp: evtl. arbeiten Sie mit Farben.

12. **Hamiltonkreis** (? Punkte)
Gegeben sei nachfolgender Graph:



(a) Wie viele Hamiltonkreise gibt es ?

(b) Wie viele Wege gibt es von B nach D ?

13. **Hamiltonkreis** (? Punkte)
Zeigen Sie für $3 \leq n \in \mathbb{N}$:

Die Anzahl der Hamiltonkreise eines vollständigen Graphen K_n ist:

$$\frac{(n-1)!}{2}$$

Zeigen Sie die Aussage zunächst explizit für den K_3 und den K_4.

Beachten Sie: Die Hamiltonkreise werden nicht als unterschiedlich betrachtet, wenn nur die Startecke unterschiedlich ist oder die Richtung beim durchlaufen.

14. **Hamiltonkreis** (? Punkte)
In der Druckerei Müller werden sechs unterschiedliche Farben der Reihe nach in einer Druckmaschine verwendet. Die Maschine muß vor jedem Farbenwechsel gereinigt werden, wobei die Reinigungszeit von den beiden aufeinander folgenden Farben abhängig ist. Die Druckerei möchte für den 6-Farbendruck eine Reihenfolge ermitteln, beginnend mit Blau, so dass die für die Reinigung benötigte Gesamtzeit minimal ist.

Die Reinigungsdauern (in Minuten) sind in der folgenden (symmetrischen) Tabelle angegeben:

Sorte	Blau	Schwarz	Gelb	Rot	Gold	Grün
Blau	0	3	12	10	10	6
Schwarz	3	0	17	15	15	20
Gelb	12	17	0	5	5	19
Rot	10	15	5	0	15	18
Gold	10	15	5	15	0	11
Grün	6	20	19	18	11	0

Zeichnen Sie die Tabelle als vollständigen bewerteten Graphen, wobei die Farben die Namen der Ecken sind und die Kanten mit den Reinigungszeiten bewertet werden.

Führen Sie den Algorithmus der Einfügung der dichtesten Ecke für das Problem des Handlungsreisenden auf diesen vollständigen bewerteten Graphen (bis zum Abbruch!) durch.

15. **Hamiltonkreis** (?? Punkte)
Professor Erdös wurde nach Graphistan zu einem Forschungsaufenthalt eingeladen. Allerdings soll er in Eulerton, New Hamilton, Oberdösach, Brookshire, Königsberg und Bergen je einen Vortrag halten. Um von einem Vortragsort zum anderen zu gelangen fliegt er mit dem Flugzeug, denn zum Glück besitzt jeder dieser Orte einen Flughafen. Er ist nun, der Umwelt zuliebe, bemüht die dabei anfallenden Reisekilometer zu minimieren. Die Entfernung (in km) sind in der folgenden (symmetrischen) Tabelle angegeben:

Ort	Euler	Hamilton	Dösbach	Shire	Königsberg	Bergen
Euler	0	500	670	320	350	410
Hamilton	500	0	720	290	470	800
Dösbach	670	720	0	890	950	920
Shire	320	290	890	0	310	550
Königsberg	350	470	950	310	0	250
Bergen	410	800	920	550	250	0

Zeichnen Sie die Tabelle als vollständigen bewerteten Graphen, wobei die Anfangsbuchstaben die Namen der Ecken sind und die Kanten mit den Entfernungen bewertet werden.

Führen Sie den Algorithmus der Einfügung der dichtesten Ecke für das Problem des Handlungsreisenden auf diesen vollständigen bewerteten Graphen (bis zum Abbruch!) mit der Startecke **H**amilton durch.

16. **Würfelgraph** (? Punkte)
Sei

$$V_n := \{v = (x_1, x_2, \ldots, x_n) \in \mathbb{B}^n \| x_i \in \{0,1\}, 1 \leq i \leq n\}$$

und $E_n := \{vu \| v, u \in V_n, x$ und y unterscheiden sich in genau 1 Koordinate$\}$ und $W_n = (V_n, E_n)$ der n-dimensionale „Würfelgraph“ (siehe Abbildung 5.3).

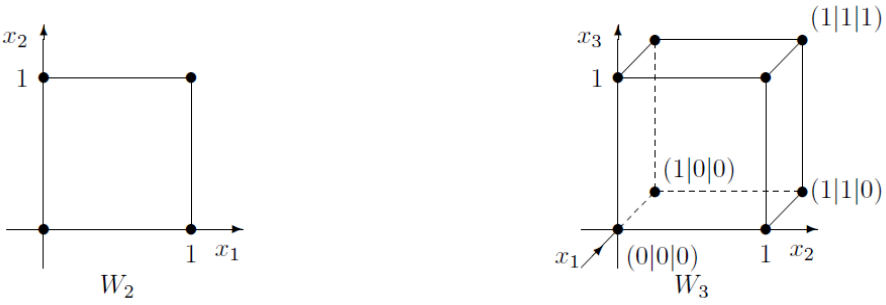


Abbildung 5.3: Würfelgraph

- Für welche $n \in \mathbb{N}$ ist W_n hamiltonsch ? Begründung/Lösungsweg nicht vergessen!
- Für welche $n \in \mathbb{N}$ ist W_n planar ? Begründung/Lösungsweg nicht vergessen!

Kapitel 6

Petri-Netze

Petri-Netze werden angewendet, um z.B. die Steuerungslogik von Prozessen darzustellen, oder Koordinations- und Synchronisationsprobleme bei nebenläufigen Prozessen zu repräsentieren, oder dynamisches Systemverhalten aufzuzeigen, usw. Sie wurden 1962 durch die Dissertation über Kommunikation mit Automaten von *Carl Adam Petri* in die Literatur eingeführt und tragen deshalb seinen Namen. Diese Netze haben breiten Zuspruch gefunden, was sich durch den seit 1979 erscheinenden Petri Net Newsletter und die seit 1980 stattfindenden Konferenzen und Workshops zeigt. Auch in der Praxis finden diese Netze Zuspruch, z.B. werden im Bereich der Workflow-Systeme auf Petri-Netzen basierende Systeme angeboten.

Die Darstellung eines Systems als Petri-Netz liefert eine formale Grundlage dafür, Aussagen über das Verhalten des Systems (z.B. das Nichteintreten bestimmter unerwünschter Zustände) zu beweisen. Im Rahmen des vorliegenden Textes müssen wir uns allerdings darauf beschränken, am Beispiel darzustellen, auf welche Weise Systeme durch Petri-Netze repräsentiert werden können, sowie die für die Untersuchung solcher Netze bedeutsamen Fragen anzudeuten.

Nebenläufige Programmierung ist schwierig, da wir nicht alle möglichen Zustände überschauen können. Dazu ein Beispiel: Gegeben seien die beiden Termersetzungsregeln $R_1 : ab \to aa$ und $R_2 : ba \to bb$. Wir betrachten die möglichen Zustände einer sequentiellen, parallelen und nebenläufigen Verarbeitung nach einem Schritt. Parallel bedeutet dabei, dass in gleichzeitig bzw. gemeinsam getakteten Prozessen die Berechnung vorgenommen wird, wogegen Nebenläufig bedeutet, dass von der sequentiellen Ausführung aller Prozesse bis hin zur parallelen Ausführung aller Prozesse alle Möglichkeiten zugelassen sind.

Zunächst betrachten wir eine **Konflikt freie Situation** mit dem Wort $W := abcba$:

Sequentiell : $W \to aacba$ oder $W \to abcbb$

Parallel : $W \to aacbb$

Nebenläufig : $W \to aacba$, $W \to abcbb$ oder $W \to aacbb$

Betrachten wir nun eine **Konflikt behaftete Situation** mit dem Wort $W := abab$:

Sequentiell : $W \to aaab$ oder $W \to abbb$ oder $W \to abaa$

Parallel : $W \to aaaa$

Nebenläufig : $W \to aaab$, $W \to abbb$, $W \to abaa$ oder $W \to aaaa$

Dazu ist zu bemerken, dass im parallelen Fall ein Fehler entsteht, wenn versucht wird, Regel R_1 und Regel R_2 parallel auf den (Konflikt-)Teil aba anzuwenden. Wenn wir Glück haben, fällt jemand der Fehler auf.

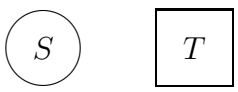
Zu der Anzahl der Möglichkeiten: Das die Zustände der nebenläufigen Verarbeitung aus den Zuständen der parallelen und sequentiellen Verarbeitung besteht, liegt an der Anzahl von zwei Regeln. Bei drei Regeln hätten wir z.B. eine andere Anzahl: Alle Möglichkeiten der sequentiellen Ausführung (3!) plus die Anzahl, dass zwei parallel ausgeführt werden und der dritte vorher oder nachher ausgeführt wird (3*2) plus die Anzahl, dass alle drei parallel ausgeführt werden, was in der Summe 13 Zustände ergibt! Bei vier Regeln sind es schon 127 mögliche Zustände.

6.1 Definition von Petri-Netzen

Ein Petri-Netz besteht aus zwei Teilen, von denen der eine als statischer Teil die logische Struktur des abzubildenden Systems modelliert, während der andere als dynamischer Teil das Verhalten des Systems im Zeitablauf wiedergibt.

6.1.1 Statischer Teil

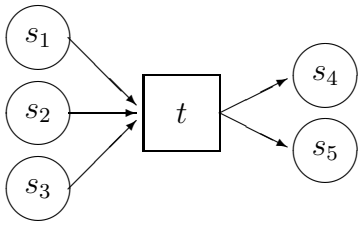
Der statische Teil eines Petri-Netzes ist ein bipartiter gerichteter Graph. Dessen Ecken werden als S-Ecken, bzw. T-Ecken bezeichnet und entsprechend der Form der Buchstaben durch Kreise, bzw. Rechtecke dargestellt:



Für eine bestimmte Ecke v heißen die Anfangsecken der gerichteten Kanten mit der Endecke v der Vorbereich $\bullet v$ von v und die Endecken der gerichteten Kanten mit der Anfangsecke v der Nachbereich $v\bullet$ von v. Meist wird der Vor-/Nachbereich von T-Ecken betrachtet, da diese bei der sogenannten Schaltung wichtig sind.

Als Motivation für die Regeln kann man sich z.B. die T-Ecken als Prozesse bzw. Threads vorstellen und die S-Ecken als die Kommunikationsports bzw. Nachrichtenspeicher. Die Kanten sind dann die möglichen Kommunikationswege.

Beispiel 6.1 *In dem folgenden Graphen hat t den Vorbereich $\bullet t = \{s_1, s_2, s_3\}$ und den Nachbereich $t\bullet = \{s_4, s_5\}$.*



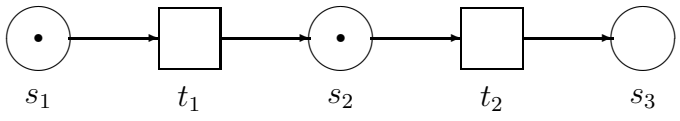
6.1.2 Dynamischer Teil

Der dynamische Teil eines Petri-Netzes besteht aus Marken. Dies sind Markierungen der S-Ecken, die sich nach festgelegten Regeln ändern.

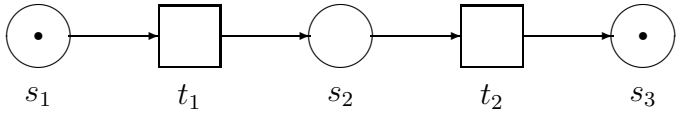
Definition 6.1 *Eine Ecke t heißt* aktiviert*, wenn jede Ecke aus $\bullet t$ eine Mindestanzahl von Marken trägt und keine Ecke aus $t\bullet$ ihre Maximalzahl von Marken trägt. Unter dem* Schalten einer aktivierten Ecke t *versteht man die Verminderung der Anzahl der Marken bei jeder Ecke aus $\bullet t$ und die Erhöhung der Anzahl der Marken bei jeder Ecke aus $t\bullet$ um jeweils eine festgelegte Anzahl.*

Wie viele Marken in den Ecken zulässig sind, welche Modifikationen beim Schalten vorgenommen werden, wann geschaltet wird usw. sollte durch die Anwendung bestimmt werden. In diesem Buch werden drei unterschiedliche Typen von Petri-Netzen vorgestellt.

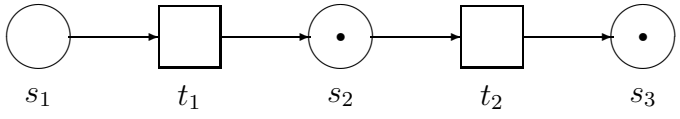
Beispiel 6.2 *Im einfachsten Fall ist das Schalten von t genau dann erlaubt (Schaltregel), wenn jede Ecke aus $\bullet t$ eine Marke trägt und keine Ecke aus $t\bullet$ eine Marke trägt. Beim Schalten verliert jede Ecke aus $\bullet t$ ihre Marke und jede Ecke aus $t\bullet$ erhält eine Marke. So kann in dem folgenden Graphen zunächst t_2 schalten*



und es ergibt sich



so dass jetzt t_1 schalten kann — mit dem Ergebnis



Anschaulich lässt sich eine Analogie der Terminologie von Petri-Netzen und z.B. Brettspielen erstellen, wie in der nachfolgenden Tabelle dargestellt.

Petri-Netze	Brettspiele
Graph und Beschriftung	Spielplan
Schaltregeln	Spielregeln
Marken	Figuren
Transitionsschaltung	Spielzug
Markendynamik	Spiel

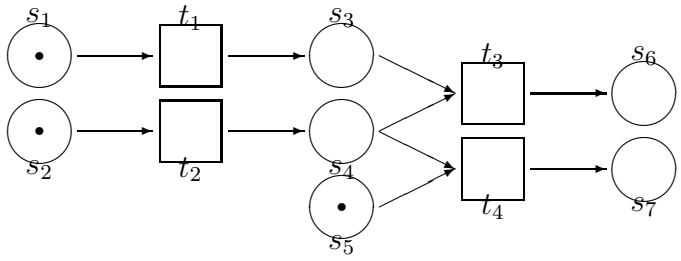
6.1.3 Konflikte und konfuse Situationen

Es kann vorkommen, dass durch die Markierungen nicht eindeutig festgelegt ist, welche T-Ecke schalten darf (was für sich genommen noch nicht schlimm ist und sich oft durch das gleichzeitige, unabhängige Schalten der dazu berechtigten T-Ecken regeln lässt) und dass durch die willkürliche Auswahl einer zu schaltenden Ecke unterschiedlich markierte Netze entstehen. Dann stehen die entsprechenden T-Ecken in *Konflikt* miteinander. Diese Situation ist u.U. aus dem Bereich der Regelsysteme bekannt: wenn gleichberechtigt mehrere Regeln gleichzeitig angewendet werden können, besteht eine Konfliktsituation. Durch die Möglichkeit der Auswahl wird der entsprechende Ablauf nicht deterministisch! Bei *konfluenten* Regeln führt der Ablauf unabhängig von der getroffenen Wahl zum gleichen bzw. zu einem äquivalenten Ergebnis. In der Praxis kommen jedoch meist nicht konfluente Regeln vor. Bei den Regeln wie auch hier bei den Petri-Netzen gilt es nun geschickte Auswahlheuristiken anzugeben, um die gewünschten Resultate zu erzielen.

Falls ein Konflikt erst dadurch entsteht, dass zwei nicht miteinander in Konflikt stehende T-Ecken in einer bestimmten zeitlichen Reihenfolge schalten, spricht man von einer *konfusen Situation*.

Beide beschriebenen Situationen können z.B. für die Analyse bestehender Geschäftsprozesse sehr interessant sein.

Beispiel 6.3 *Wenn in dem folgenden Petri-Netz die Ecken s_3, s_4, s_5 markiert und s_6, s_7 unmarkiert sind, dann stehen t_3 und t_4 in Konflikt miteinander. Wenn t_3 schaltet, wird ein Schalten von t_4 verhindert und umgekehrt, so dass je nach Auswahl der zu schaltenden T-Ecke unterschiedliche Folgezustände des Netzes entstehen. Mit der angegebenen Markierung liegt eine konfuse Situation vor, da der beschriebene Konflikt genau dann entsteht, wenn t_1 vor t_2 (oder gleichzeitig mit ihr) schaltet, während er vermieden wird, wenn t_2 vor t_1 schaltet (weil dann t_4 zu einem Zeitpunkt schalten kann, an dem t_3 noch nicht aktiviert ist).*



Petri-Netze werden danach klassifiziert, welche Arten von Markierungen erlaubt sind:

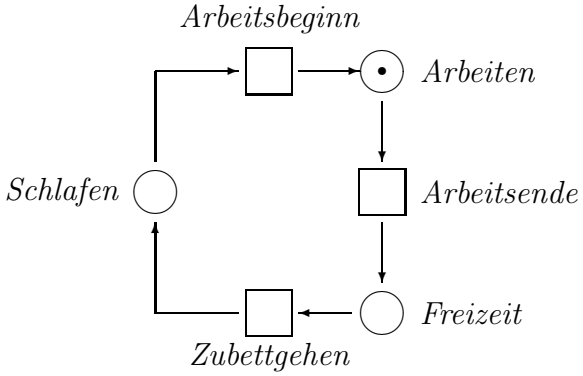
- jede Ecke darf höchstens eine Marke tragen;
- jede Ecke darf mehrere ununterscheidbare Marken tragen;
- jede Ecke darf mehrere unterscheidbare Marken tragen.

6.2 Bedingungs/Ereignis-Systeme

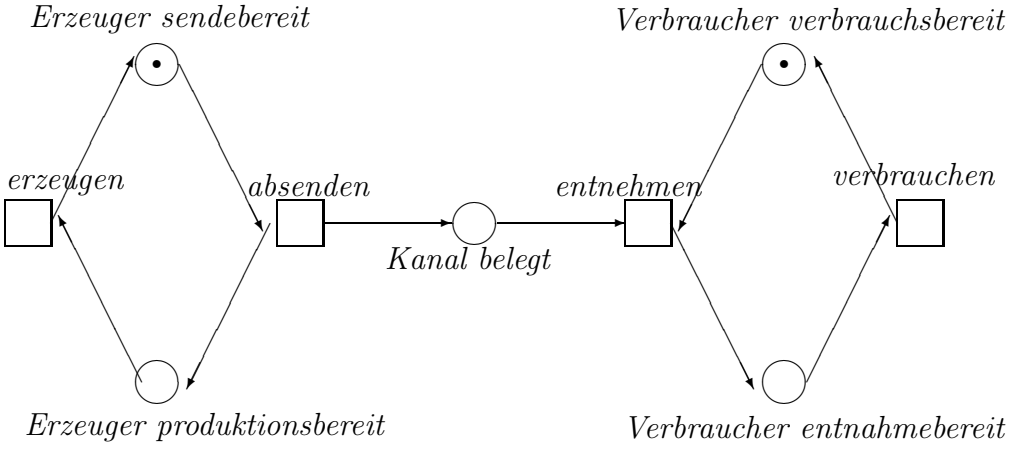
In dem einfachsten Typ von Petri-Netzen werden die S-Ecken als Bedingungen interpretiert, deren Markierung Auskunft darüber gibt, ob die jeweilige Bedingung erfüllt (=markiert) oder nicht erfüllt (=nicht markiert) ist. Dieser Interpretation entsprechend kann jede S-Ecke höchstens eine Marke tragen. Die T-Ecken stellen Ereignisse dar, die genau dann stattfinden dürfen, wenn alle Bedingungen des Vorbereichs erfüllt und alle Bedingungen des Nachbereichs nicht erfüllt sind. Das Stattfinden (=Schalten) eines Ereignisses t wird

im Netz dadurch dargestellt, dass alle Ecken aus $\bullet t$ ihre Markierungen verlieren und alle Ecken aus $t\bullet$ markiert werden.

Beispiel 6.4 *Ein vereinfachtes Modell eines Tagesablaufs unterscheidet die Phasen Schlafen, Arbeiten und Freizeit:*



Beispiel 6.5 *Im folgenden wird eine direkt am Verbraucherverhalten orientierte Produktion gezeigt.*



Definition 6.2 (Fälle und Schritte) *Die Menge aller gleichzeitig erfüllten Bedingungen eines Netzes heißt ein* Fall *dieses Netzes.*

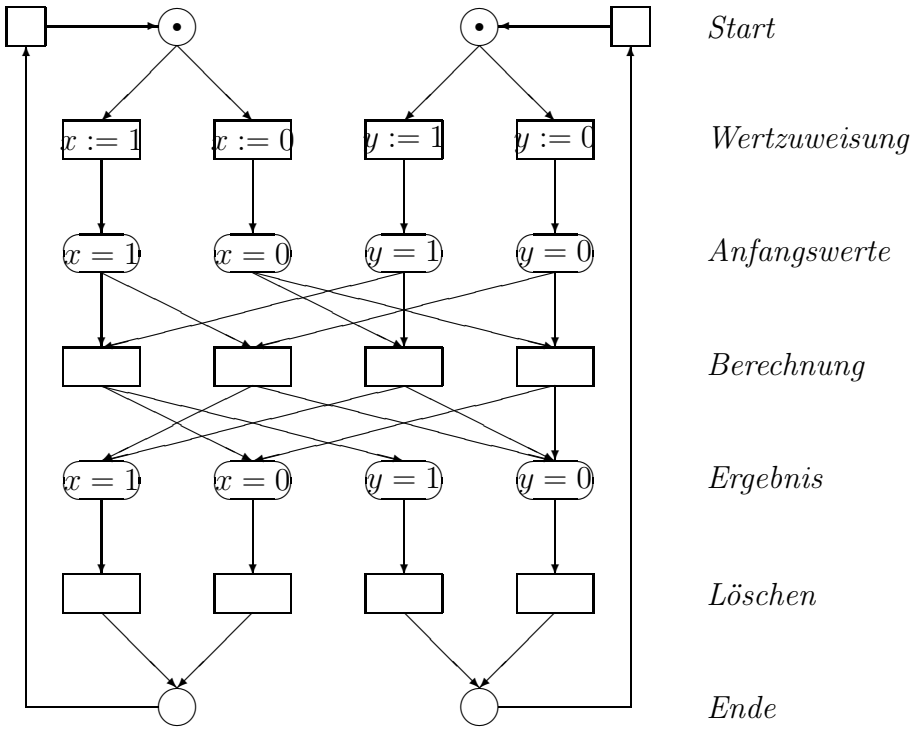
Als Schritt *bezeichnet man den Übergang von einem Fall zu einem anderen durch das Eintreten eines Ereignisses oder durch das gleichzeitige Eintreten mehrerer Ereignisse.*

Für ein Netz sind im allgemeinen nur bestimmte Fälle sinnvoll. So sind beispielsweise für das „Tageszeiten“-Netz genau diejenigen Fälle sinnvoll, in denen genau eine Bedingung erfüllt ist.

Beispiel 6.6 *Ein Halbaddierer soll die Binärziffern x und y so miteinander verknüpfen, dass anschließend x die Ziffer $x\,XOR\,y$ und y den Übertrag $x\,AND\,y$ enthält. Seine Darstellung als Petri-Netz folgt weiter unten. Dabei werden in diesem Petri-Netz die möglichen vier Fälle ($x = 1$,$y = 1$;$x = 1$,$y = 0$;etc.) erfaßt.*

Die Konflikte, die im dargestellten Fall verschiedene Schritte ermöglichen, drücken aus, dass die Werte für x und y jeweils willkürlich festgelegt werden dürfen, d.h. die T-Ecken zur Wertzuweisung können beliebig schalten. Durch die vorgenommene Anordnung kann jedoch von den T-Ecken, die z.B. die Wertezuweisung für x vornehmen, nur genau eine schalten. In der Berechnungsebene kann dann nur genau eine T-Ecke schalten. Anschließend liegen die Marken in den S-Ecken, die für die jeweiligen Werte von x und y stehen. Die berechneten Werte werden anschließend wieder gelöscht, damit der Halbaddierer wieder für eine neue Berechnung zur Verfügung steht.

Von dem eingezeichneten Fall aus kann ein Schritt etwa im Schalten der T-Ecken $x := 1$ und $y := 0$ bestehen. Es wird dann die zweite T-Ecke von links in der Berechnungsebene aktiviert, die dann die Marke in die S-Ecken $x = 1$ und $y = 0$ durchschaltet.



Wenn ein reales System als Petri-Netz dargestellt wird, geschieht das häufig mit dem Ziel, das System auf gewisse Eigenschaften hin zu untersuchen. Bei Bedingungs/Ereignis-Systemen sind solche Eigenschaften beispielsweise in nachfolgender Definition aufgeführt.

Definition 6.3 (Zyklische und lebendige Netze) *Ein Bedingungs/Ereignis-System heißt* zyklisch*, falls ausgehend von einem beliebigen (sinnvollen) Fall jeder andere (sinnvolle) Fall durch endlich viele Schritte erreicht werden kann.*

Ein Bedingungs/Ereignis-System heißt lebendig*, falls ausgehend von einem beliebigen (sinnvollen) Fall für ein beliebiges Ereignis t ein Fall herbeigeführt werden kann, so dass t aktiviert ist.*

Der/die LeserIn möge sich überlegen, dass jedes zyklische Netz auch lebendig ist; jedes lebendige Netz aber nicht unbedingt zyklisch.

Sowohl der Tagesablauf als auch der Halbaddierer sind zyklisch und lebendig.

Aufgaben

1. Überlegen Sie sich, dass man einen endlichen Automaten als Bedingungs/Ereignis-System mit einelementigen Fällen auffassen kann.

2. Modellieren Sie die Vorbereitung eines Ausflugs als Bedingungs/Ereignis-System. Beachten Sie dabei folgende Notwendigkeiten:

   (a) Zunächst ist das Reiseziel zu bestimmen.

   (b) Danach ist das Verkehrsmittel, Pkw oder Bahn, auszuwählen.

   (c) Parallel zu (b) darf der benötigte Proviant eingekauft werden.

   (d) Nach (b) und (c) wird entschieden, welches Gepäck sonst noch mitgenommen werden soll.

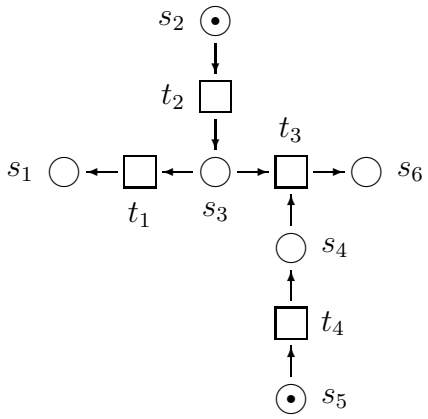
   (e) Im Fall einer Autofahrt ist das Auto aufzutanken; im Fall einer Bahnfahrt sind Fahrkarten zu kaufen.

   (f) Wenn alles dies erledigt ist, kann es losgehen.

3. Als Fallgraphen eines Bedingungs/Ereignis-Systems bezeichnet man den gerichteten Graphen $G(V, E)$, dessen Ecken die (sinnvollen) Fälle des Bedingungs/Ereignis-Systems sind. Eine Kante verläuft dabei jeweils von einem Fall v_i zu einem Fall v_j, der von v_i aus durch *einen* Schritt herbeigeführt werden kann. Zeichnen Sie die Fallgraphen für den Tagesablauf und den Halbaddierer, und weisen Sie nach, dass diese beiden Graphen

stark zusammenhängend sind. (Dies beweist, dass die jeweiligen Petri-Netze zyklisch sind.)

4. Wieso liegt in dem folgenden Petri-Netz eine konfuse Situation vor?



5. Bilden Sie den folgenden Ablauf in einer Kantine in einem Bedingung-Ereignis-System ab!

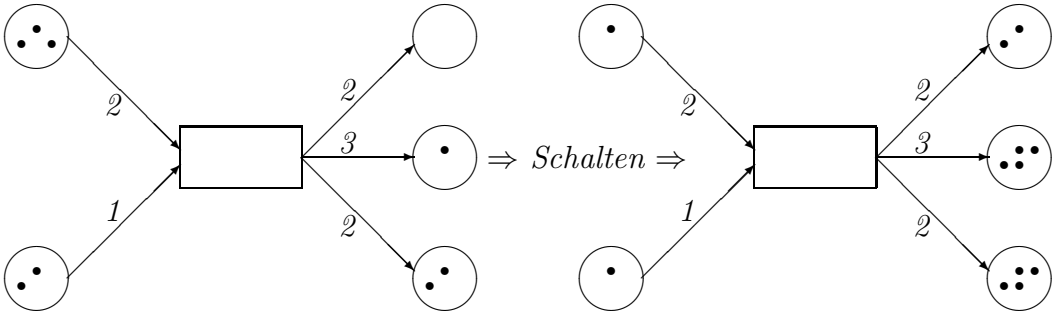
   - Neue Kunden kommen durch eine Eingangstür (Systemgrenze, die als markenerzeugende Transition abgebildet werden kann).
   - Die Kunden entscheiden sich für eines von zwei Gerichten, für die jeweils eine eigene Warteschlange existiert.
   - Die zwei Ausgaben für die Gerichte werden durch Bedienstete betreut, die je nach Andrang an den einzelnen Ausgaben tätig werden.
   - Nach Empfang des Hauptgerichts reihen sich die Kunden in eine einzige Kassenwarteschlange ein.
   - Nach der Kasse nehmen die Kunden ihre Mahlzeit in einem Essensraum ein.
   - Nach Essen verlassen sie die Kantine (Systemgrenze, wie Eingang).

6.3 Stellen/Transitionen-Systeme

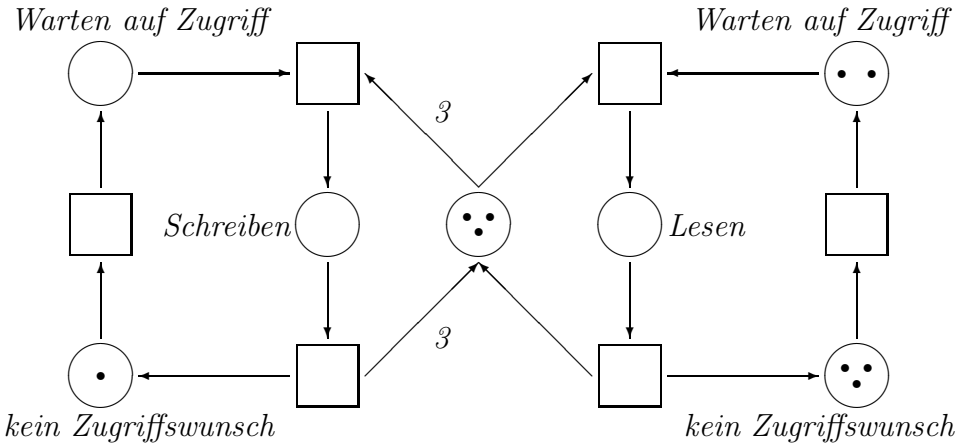
Weitere Vorgänge können als Petri-Netz modelliert werden, wenn jeder Ecke s_i eine gewisse Zahl von Marken, die zwischen 0 und k_i – der für s_i festgelegten Kapazität – liegt, zugewiesen werden darf. Man spricht dann von Stellen/Transitionen-Systemen und bezeichnet die S-Ecken als Stellen und die

T-Ecken als Transitionen. Als zusätzliche Erweiterung darf auch festgelegt werden, dass beim Schalten einer Transition t Ecken aus $\bullet t$ [bzw. $t\bullet$] mehr als eine Marke verlieren [bzw. gewinnen]. In diesem Fall erhält die betreffende Kante $s_i t_j$ [bzw. $t_i s_j$] ein Gewicht w_{ij}, das angibt, um wie viele Marken sich die Markierung der Stelle ändert, wenn die Transition schaltet.

Beispiel 6.7 *In diesem Beispiel soll die Schaltregel eines Stellen/Transitions-Netzes verdeutlicht werden. Dabei stellt die linke Seite die Situation vor dem Schalten dar und die rechte die Situation nach dem Schalten. Die Zahlen an den Kanten geben an, wie viele Marken benötigt werden zum Schalten bzw. wie viele nach dem Schalten verteilt werden.*



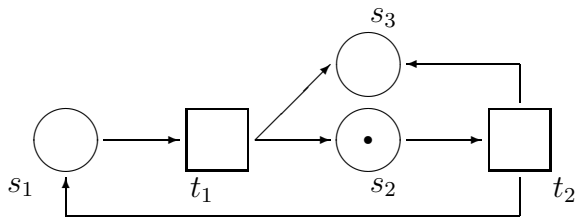
Beispiel 6.8 *Der Zugriff auf einen Datenbestand ist entweder bis zu drei (von insgesamt fünf) lesenden Prozessen oder einem schreibenden Prozeß erlaubt. Die Koordination dieser Prozesse kann in folgender Weise durch ein Petri-Netz dargestellt werden. (Alle nicht ausdrücklich gekennzeichneten Kanten haben das Gewicht 1, alle Kapazitäten sind beliebig groß.)*



Definition 6.4 (Sicherheit) *Ein Petri-Netz heißt* sicher *bezüglich einer bestimmten Anfangsmarkierung, wenn für jede S-Ecke die Anzahl der Marken*

nach beliebig häufigem Schalten der T-Ecken (ohne Beachtung irgendwelcher Kapazitäten) beschränkt bleibt.

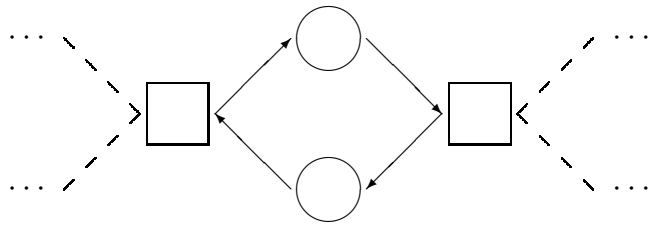
Beispiel 6.9 *Dieses Petri-Netz ist nicht sicher bzgl. der angegebenen Markierung, da nach hinreichend häufigem Schalten von t_1 und t_2 die Anzahl der Markierungen von s_3 beliebig groß wird.*



Definition 6.5 (Deadlocks und Traps) *Ein* Deadlock *eines Petri-Netzes ist eine Menge von Stellen, die, wenn sie zu einem gewissen Zeitpunkt keine Marken mehr tragen, auch in der Folgezeit nie wieder markiert werden.*

Ein Trap *eines Petri-Netzes ist eine Menge von Stellen, die, wenn mindestens eine von ihnen mindestens eine Marke enthält, auch in der Folgezeit nicht alle gleichzeitig unmarkiert sein können.*

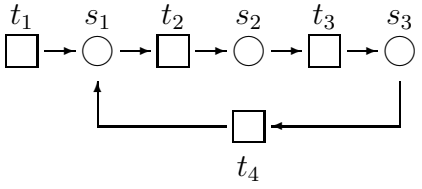
Beispiel 6.10 *Die Stellen in dem folgenden Ausschnitt aus einem Petri-Netz sind sowohl ein Deadlock (ohne Marke) als auch ein Trap (mit einer Marke).*



Aufgaben

1. Eine Studienordnung sieht (u.a.) vor, dass während des Grundstudiums vier bei Nichtbestehen beliebig oft wiederholbare Prüfungen abgelegt werden müssen. Modellieren Sie das Ablegen dieser Prüfungen als Stellen/Transitionen-System. Dieses Petrinetz soll genau zwei S-Ecken besitzen, wobei die Anzahl der Marken in der einen S-Ecke angibt, wie viele Prüfungen noch zu absolvieren sind und die Anzahl der Marken in der anderen S-Ecke angibt, wie viele Prüfungen bereits bestanden wurden.

2. Ist in diesem Stellen/Transitionen-System die Menge der S-Ecken ein Deadlock oder ein Trap? Es seien zu Anfang alle S-Ecken unmarkiert. Ist das System bezüglich dieser Anfangsmarkierung sicher?



3. Transformieren Sie das Bedingung-Ereignis-Netz aus der Aufgabe 5 (Seite 153) in ein Stellen-Transitions-System mit folgenden Einschränkungen:

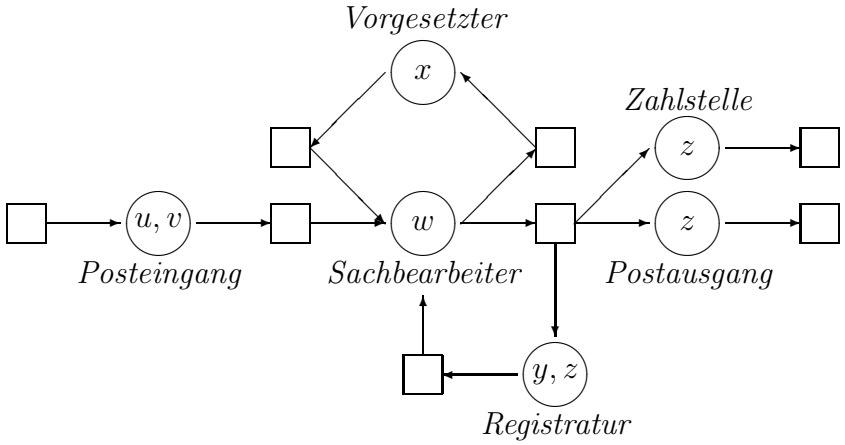
   - Die Warteschlangen für die Gerichte umfassen je maximal 10 Kunden.
   - Es gibt nur eine Bedienstete.
   - Die Kassenwarteschlange besitzt eine Kapazität von 15 Kunden.
   - Im Essensraum gibt es 150 Plätze.

6.4 Prädikat/Ereignis-Systeme

Eine weitere Ausdehnung der Modellierungsmöglichkeiten mit Petri-Netzen ergibt sich schließlich, wenn die einzelnen Marken einer S-Ecke als unterscheidbar angesehen werden. Das Schalten von T-Ecken wird dabei wiederum als Ereignis bezeichnet. Die S-Ecken werden jetzt als Prädikate interpretiert, die genau von denjenigen Objekten erfüllt werden, die durch die dort vorhandenen Marken repräsentiert werden.

Beispiel 6.11 *In einem BAföG-Amt kommen Anträge in der Poststelle an und werden dann einzelnen Sachbearbeitern zur Bearbeitung zugewiesen. Diese können in Zweifelsfällen bei ihrem Vorgesetzten Rücksprache nehmen. Wenn die Bearbeitung abgeschlossen ist, veranlassen sie*

- *den Versand eines Bescheides an den Antragsteller,*
- *eine Mitteilung an die Zahlstelle sowie*
- *die Ablage der Akte in der Registratur*



Die S-Ecken werden jeweils als Prädikate über der Menge aller BAföG-Antragsteller interpretiert, wobei „. . . “ als Platzhalter (Parameter) des Prädikates verwendet wird.

- *Der Antrag . . . wartet im Posteingang.*
- *Die Akte . . . befindet sich in Bearbeitung.*
- *Die Akte . . . liegt dem Vorgesetzten vor.*
- *Der Bescheid . . . ist fertig zum Absenden.*
- *Der . . . zustehende Betrag wird angewiesen.*
- *Die Akte . . . liegt in der Registratur.*

Aus jedem Prädikat wird eine wahre Aussage, wenn für „. . . “ der Name einer Marke der entsprechenden Ecke eingesetzt wird. Beim Schalten tragen die betroffenen Marken alle den selben Namen.

Wie bei Stellen/Transitionen-Systemen kann auch hier die endliche Leistungsfähigkeit der S-Ecken durch die Angabe von Kapazitäten modelliert werden.

Aufgaben

1. Stellen Sie das Studium an Ihrer Hochschule als Prädikat/Ereignis-System dar. Berücksichtigen Sie dabei folgende Prädikate:

   (a) Eine Person bewirbt sich um einen Studienplatz. (Die Bewerbung kann abgelehnt oder angenommen werden. Im Falle der Annahme ist

die Zulassung zum Grund- oder zum Hauptstudium möglich. Im Fall der Ablehnung kann die Person die Bewerbung wiederholen bzw. aufrechterhalten oder die Bemühung um einen Hochschulabschluß aufgeben.)

(b) Eine Person befindet sich im Grundstudium.

(c) Eine Person befindet sich im Hauptstudium.

(d) Eine Person hat die Bemühung um einen Hochschulabschluß aufgegeben.

(e) Eine Person hat die Hochschule am Ende des Hauptstudiums mit bestandener Abschlußprüfung verlassen.

2. **Modellierung** (9 Punkte)
Im nachfolgenden werden Definitionen von Petri-Netzklassen aufgeführt. Ordnen Sie die Netzwerke aus den Abbildungen 6.1 (Seite 159) und 6.2 (Seite 159) diesen Definitionen zu. Erläutern Sie warum ggf. dass Netz zu der Klasse dazu gehört bzw. warum es nicht dazu gehört.

(a) Zustandsmaschinen (ZM) haben ausschließlich unverzweigte Transitionen, die außerdem nicht am absoluten Rand liegen: Eine **Zustandsmaschine** ist ein Netz, bei dem jede Transition der Vorbereich und der Nachbereich nur genau eine Stelle beinhalten.

(b) Synchronisationsgraphen (SG) haben ausschließlich unverzweigte Stellen, die außerdem nicht am absoluten Rand liegen: Ein **Synchronisationsgraph** ist ein Netz, bei dem jede Stelle der Vorbereich und der Nachbereich nur genau eine Transition beinhalten.

(c) Bei verallgemeinerten Zustandsmaschinen (VZM) bzw. Synchronisationsgraphen (VSG) sind absolute Randknoten beider Arten zugelassen: Eine **verallgemeinerte Zustandsmaschine** ist ein Netz, bei dem jede Transition der Vorbereich und der Nachbereich maximal eine Stelle beinhalten.

(d) Ein **verallgemeinerter Synchronisationsgraph** ist ein Netz, bei dem jede Stelle der Vorbereich und der Nachbereich maximal eine Transition beinhalten.

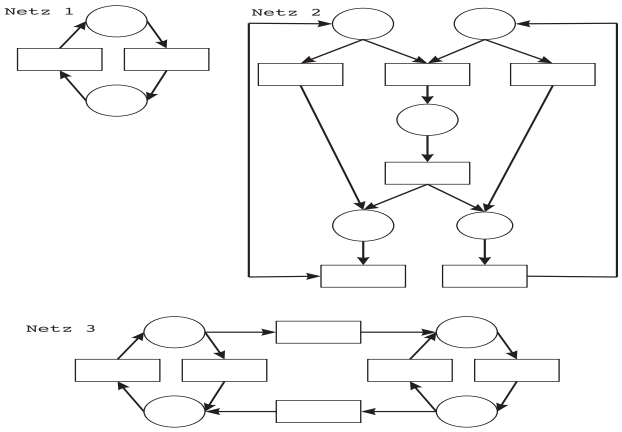


Abbildung 6.1: Trennende Beispiele für Petri-Netzklassen I

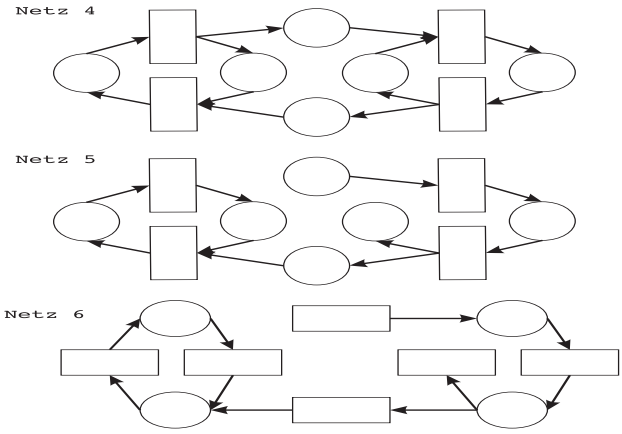


Abbildung 6.2: Trennende Beispiele für Petri-Netzklassen II

Kapitel 7

Graphtransformationen

Wenn man den Begriff *Grammatik* liest oder hört, assoziiert man damit i.allg. die Beschreibung des Aufbaus einer Gegenwartssprache, wie etwa Englisch oder Deutsch. Diese stellt, ausgehend von den Grundeinheiten Wort und Satz, alle sprachlichen Erscheinungen (z.B. Bau der Sätze, Laute oder Wortbildungen) exakt und übersichtlich dar. Im Bereich der Informatik, speziell der formalen Sprachen, assoziiert man schon etwas spezieller mit diesem Begriff eine formale Definition von *Zeichenkettengrammatiken*. Die wenigsten denken jedoch bei dem Begriff Grammatik an die sogenannten *Graphgrammatiken*, die nicht nur über Zeichenketten operieren, sondern über beliebigen Graphen. Es wird daher zwischen Graphgrammatiken und *Zeichenkettengrammatiken* bzw. *String-Grammatiken* unterschieden.

In den letzten 25 Jahren wurden verschiedene Ansätze für die Graphgrammtiken bzw. Graphtransformationen entwickelt, die sich teilweise erheblich voneinander unterscheiden. Jeder von ihnen eignet sich besonders für ein spezielles Anwendungsgebiet. Auf der einen Seite wurde versucht, Graphgrammatiken für Software-Engineering einzusetzen (z.B. das System PROGRES). Diese Ansätze zeichnen sich dadurch aus, daß sie neben einem graphischen Kern auch über eine nicht graphische Komponente verfügen. Dem gegenüber stehen mehr theoretisch orientierte Ansätze. Sie beschäftigen sich z.B. mit der Übertragung der Theorie Formaler Sprachen auf mehrdimensionale Strukturen, der logischen Spezifikation von Graphsprachen, sowie der effizienten Implementierung von Ersetzungssystemen[1].

In diesem Buch wird ein aus der Literatur bekannter allgemeiner Ansatz zur Beschreibung von Graphgrammatiken vorgestellt, der zum einen die grund-

[1] Einen guten Überblick geben die Veröffentlichungen: Kreowski, H.-J.; Rozenberg, G.: *On Structured Graph Grammars. I, II*, in: Information Sciences, Vol. 52, pp. 185-210, 221-246, 1990

legenden Merkmale von Graphgrammatiken verdeutlichen und zum anderen Unterschiede bzw. Gemeinsamkeiten der in der Literatur verwendeten Graphgrammatiken aufzeigen soll.

Um nun diese allgemeinere Graphgrammatik definieren zu können, muß zunächst die Definition eines Graphen verallgemeinert bzw. *angepasst* werden. Σ_V und Σ_E seien dabei zwei Alphabete.

Definition 7.1 *Ein gerichteter und Kanten und Ecken beschrifteter* Graph G *ist ein 6-Tupel* $G = (V, E, s, t, l, m)$*, wobei*

V $\neq \emptyset$ *eine endliche Menge von Ecken ist,*

E $\subseteq V \times V$ *eine Menge von Kanten ist,*

s,t : $E \to V$ *Funktionen sind, die jeder Kante* $e \in E$ *ihre Quell- bzw. Zielecke zuordnet,*

l : $V \to \Sigma_V$ *eine Sortenzuordnungsfunktion für Ecken ist und*

m : $E \to \Sigma_E$ *eine Sortenzuordnungsfunktion für Kanten ist.*

Der leere Graph G mit $V_G = \emptyset$ wird mit λ bezeichnet. Ein Graphhomomorphismus $g : G \to G$‘ ist eine Zuordnung zwischen den Mengen von Ecken bzw. Kanten der beiden Graphen, so daß Inzidenzen und Beschriftungen erhalten bleiben. Ist die Zuordnung zwischen den Mengen bijektiv, so heißt g auch Graphisomorphismus.

Seien $K \subseteq G$ und $L \subseteq G$‘ Graphen bzw. Teilgraphen und $g : K \to L$ ein Graphhomomorphismus, mit $g(K) = L$. Dann können die beiden Graphen G und G‘ mit den beiden *Klebepunkten* K und L gemäß g zu einem neuen Graphen G“ *verklebt* werden. Für $K = L = \lambda$ erhält man z.B. die disjunkte Vereinigung der beiden Graphen.

Die nachfolgenden Definitionen legen nun die Struktur der allgemeinen Graphgrammatik fest.

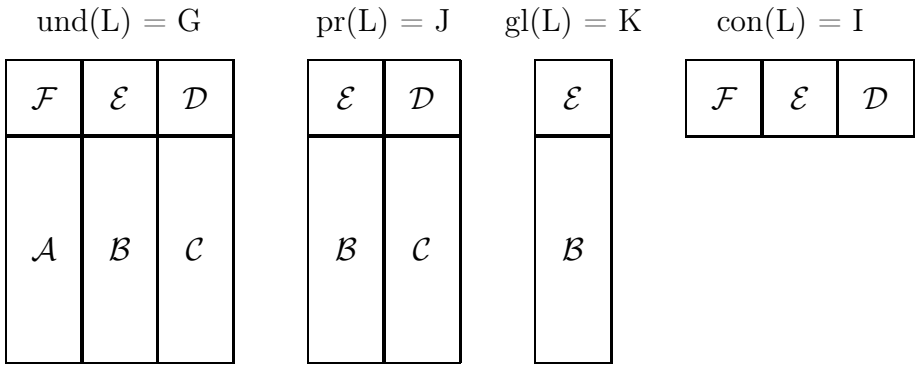
Definition 7.2 *Sei* G *ein Graph.*

- *Ein Paar* $R = (G, K)$ *heißt* schwach strukturierter Graph*, falls* $K \subseteq G$ *ein Teilgraph von* G *ist. Als Notation für die Teile von* R *wird festgelegt, daß* $G = und(R)$ *und* $K = gl(R)$*.*
- *Ein 4-Tupel* $L = (G, I, J, K)$ *heißt* strukturierter Graph*, falls* I *ein Teilgraph von* G *ist und* $K \subseteq J \subseteq G$ *Teilgraphen sind. Neben* $und(L)$ *und* $gl(L)$ *werden als Notation*[2] *festgelegt, daß* $I = con(L)$ *und* $J = pr(L)$*.*

[2]Die Notationen rühren von folgenden englischen Ausdrücken her: *und*erlying structure, *gl*uing part, *con*tact part, *pr*otected part, gluing ad*apt*or.

- *Eine* strukturierte Produktion *ist ein 4-Tupel* $p = (L, R, apt, C)$*, wobei*
  - L *ein strukturierter Graph, die linke Seite von* p *ist,*
  - R *ein schwach strukturierter Graph, die rechte Seite von* p *ist,*
  - $apt:$ $gl(R) \rightarrow gl(L)$*, ein Isomorphismus, die* Kleberelation *ist und*
  - C $= (C_{in}, C_{out})$ *ein geordnetes Paar der beiden Relationen ist, genannt* Verknüpfung*, mit* $C_{in}, C_{out} \subseteq \Sigma_V \times \Sigma_E \times \Sigma_V \times \Sigma_E \times \Sigma_V$.

In der nachfolgenden Abbildung ist ein strukturierter Graph $L = (G, I, J, K)$ und die Zerlegung in seine einzelnen Teilgraphen skizziert. Der Teilgraph $pr(L)$ wird sich bei der Anwendung einer Produktion als der *bewahrende* Teil eines strukturierten Graphen herausstellen, d.h. bei einer Produktionsanwendung wird dieser Teil nicht modifiziert.



Die Anwendung einer Produktion $p = (L, R, apt, C)$ auf einen Graphen M wird bei diesem Ansatz in die folgenden fünf Schritte unterteilt. Zum besseren Verständnis sind diese Schritte in der nächsten Abbildung visualisiert.

1. *Choose*: Suche einen Graphhomomorphismus $g : und(L) \rightarrow M$, d.h. suche in M ein Vorkommen der linken Seite von p.

2. *Check*: Überprüfe die Anwendungsbedingungen, die *gültiges* bzw. *ungültiges* Vorkommen der linken Seite in M unterscheiden. Im allg. ist die Produktion p anwendbar, d.h. das Vorkommen gültig, falls gilt: Wenn $a \in V_{g(und(L))}$ mit einer Kante $e \in E_M \setminus E_{und(L)}$ inzident ist, dann ist $a \in V_{g(con(L))}$.

3. *Remove*: Entferne verschiedene Teile des in Schritt eins gefundenen und in Schritt zwei als gültig geprüften Subgraphen in M. Sei M‘ dieser Restgraph, dann gilt, daß

   $$V_{M'} = V_M \setminus (V_{g(und(L))} \setminus V_{g(pr(L))}) \text{ und}$$

$$E_{M'} = \{e \in E_M \setminus (E_{g(und(L))} \setminus E_{g(pr(L))}) \| s_M(e), t_M(e) \in V_{M'}\}.$$

4. *Add*: Addiere zu dem restlichen Teil von M, d.h. M', die rechte Seite der Produktion p. Dies wird hier unter Verwendung der Kleberelation f basierend auf *apt* durchgeführt. f wird definiert durch $f : gl(R) \to g(gl(L))$, mit

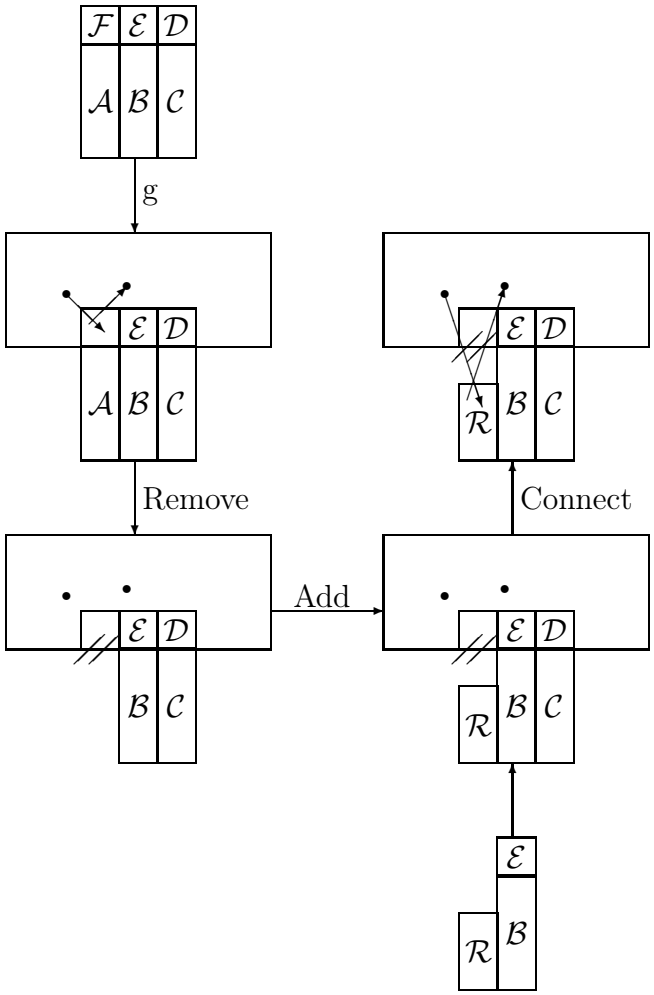
   $$\forall a \in V_{gl(R)} : f_V(a) = g_V(apt_V(a)) \text{ und}$$
   $$\forall e \in E_{gl(R)} : f_E(e) = g_E(apt_E(e)).$$

5. *Connect*: Verbinde M' mit der rechten Seite der Produktion p (Einbettung). Diese Einbettung wird gemäß $C = (C_{in}, C_{out})$ vorgenommen.

   - Für jede in Schritt drei entfernte Kante $e \in E_M$, mit $s_M(e) \in V_M \setminus V_{g(und(L))}$ und $t_M(e) \in V_{g(und(L))} \setminus V_{g(pr(L))}$ und für jede Ecke $a \in V_{und(R)} \setminus V_{gl(R)}$ wird eine neue Kante von $s_M(e)$ nach a mit der Sorte s gezogen,
     falls $(l_{und(R)}(a), s, l_M(s_M(e)), m_M(e), l_M(t_M(e))) \in C_{in}$.
   - Für jede in Schritt drei entfernte Kante $e \in E_M$, mit $t_M(e) \in V_M \setminus V_{g(und(L))}$ und $s_M(e) \in V_{g(und(L))} \setminus V_{g(pr(L))}$ und für jede Ecke $a \in V_{und(R)} \setminus V_{gl(R)}$ wird eine neue Kante von a nach $t_M(e)$ mit der Sorte s gezogen,
     falls $(l_{und(R)}(a), s, l_M(t_M(e)), m_M(e), l_M(s_M(e))) \in C_{out}$.

Die Teilgraphen $gl(L)$ und $gl(R)$ werden als *Klebepunkte* bei einer Produktionsanwendung verwendet. Sie sind damit Bestandteil der Einbettungsfunktion, d.h. die rechte Seite einer Produktion kann alleine über die Klebepunkte eingebettet werden.

In der folgenden Abbildung ist die Vorgehensweise dargestellt. Im linken oberen Teil des Bildes ist die linke Seite dargestellt. Mittels der Funktion g wurde ein isomorpher Graph in einem Graphen gefunden (dem Pfeil mit Beschriftung g folgend). In diesem Graphen sind Beispielhaft zwei Kanten zwischen diesem Teilgraphen und seiner Umgebung eingezeichnet. Die Operation Remove entfernt nun Teile des Graphen (dem Pfeil mit Beschriftung Remove folgend), wobei aus technischen Gründen eine Linie im Bild durch Durchstreichen sich wegzudenken ist. Angedeutet sind die Ecken der Umgebung. Nun fügt die Operation Add die rechte Seite der Produktion ein (den Pfeilen folgend). Im letzten Schritt wird nun gemäß Einbettungsfunktion eine Verknüpfung von bestimmten Ecken der rechten Seite mit der Umgebung, nämlich maximal den Ecken, die vorher mit der linken Seite verbunden waren, vorgenommen.



Die Einbettungsfunktion, realisiert mit der Verknüpfung C und besagten Klebepunkten, wird hier rein über die Sorten definiert. In Schritt zwei (*Check*) bzw. auch in Schritt eins (*Choose*) können zwar neben oder anstelle der dort aufgeführten Restriktionen weitere Einschränkungen angegeben werden, doch wird die Einbettung letztendlich *nur* über Sorten definiert.

Kanten können nur zwischen Ecken des *umliegenden* Graphen (M‘) und Ecken des *neuen* Teils der rechten Seite ($und(R) \setminus gl(R)$) gezogen werden, wenn zu den Ecken des umliegenden Graphen vorher Kanten bestanden, die durch die Produktionsanwendung gelöscht wurden. Darüber hinaus muß die Richtung der Kanten beibehalten werden, d.h. eine Ecke im umliegenden Graphen, die vor der Produktionsanwendung Quellecke einer gelöschten Kante war, muß es auch nach der Produktionsanwendung sein, sofern eine neue Kante durch die Verknüpfung C zu ihr hingezogen wird. Es ist aber auch zulässig, Kanten zu löschen bzw. für gelöschte Kanten keine neuen Kanten von der

Einbettungsfunktion generieren zu lassen.

7.1 Destruktive Ansätze

Für die folgenden Graphgrammatiken ist für eine Produktion $p = (L, R, apt, C)$ der *bewahrende* Teil $pr(L) = \lambda$, d.h. bei der Anwendung einer Produktion wird die linke Seite dieser Produktion komplett entfernt. Konsequenterweise gilt dann für die sogenannten *Klebepunkte* $gl(L) = \lambda = gl(R)$ und die Kleberelation apt ist die Identität auf λ, dem leeren Graphen.

Die Verknüpfungsrelation C erhält bei diesen Formalismen eine besondere Bedeutung, da die Einbettungsfunktion *nur* über sie definiert wird. Will man keine nichtzusammenhängende Graphen in einem Ableitungsschritt erzeugen, muß sichergestellt sein, daß die Einbettungsfunktion nicht leer ist. Im Falle einer Produktionsanwendung muß dann für die gelöschten Kanten mindestens eine neue Kante von C erzeugt werden.

7.1.1 Directed Node-Label-Controlled

Die Produktionen der *Directed Node-Label-Controlled Graph Grammars* (DNLC), eine Klasse von *Node-Label-Controlled Graph Grammars* (NLC), zeichnen sich zunächst dadurch aus, daß die linke Seite ihrer Produktionen nur aus einer Ecke bzw. einem strukturierten Graphen der Form $L = (\hat{l}, \hat{l}, \lambda, \lambda)$ besteht, wobei $\hat{l}$ ein Graph mit nur einer Ecke sei. Die linke Seite einer Produktion ist dann vollständig durch eine Sorte beschrieben. DNLC's gehören damit der Klasse der 1-NLC's an.

Die Kanten bei allen NLC Graphgrammatiken haben keine Sorten bzw. nur genau eine. Es ist also $\Sigma_E = \{*\}$, wobei $*$ eine beliebige Sorte sei. Die Einträge in der Verknüpfungsrelation C sind dann von der Form $(a, *, b, *, l)$, mit $a, b, l \in \Sigma_V$. Wenn nun bei einer Produktionsanwendung eine Ecke der Sorte b mit der linken Seite der Produktion über eine Kante verbunden war, wird diese Ecke mit *allen* durch die rechte Seite der Produktion neu hinzugefügten Ecken der Sorte a verbunden.

Auch andere Typen von NLC Graphgrammatiken lassen sich mit dem angegebenen Formalismus beschreiben. Bei der Verwendung von ungerichteten Graphen sind nur kleine Modifikationen der Definitionen vorzunehmen.

Bei einem zu beachtenden NLC Ansatz, den *Graph Grammars with Neighbourhood-Controlled Embedding* (NCE), gibt es aber größere Schwierigkeiten. Hier muß die Verknüpfungsrelation C statt über Sorten teilweise über Ecken definiert werden, etwa der Form $C_{in} \subseteq V_R \times \Sigma_E \times \Sigma_V \times \Sigma_E \times V_L$, wobei V_R bzw. V_L

jeweils die Ecken der rechten bzw. linken Seite einer Produktion sind. Bei NCE Graphgrammatiken wird also bei der Einbettung nur eine Kante von einer Ecke einer bestimmten Sorte des umliegenden Graphen zu einer ganz bestimmten Ecke der rechten Seite gezogen, wenn vorher eine Ecke dieser Sorte des umliegenden Graphen zu einer ganz bestimmten Ecke der linken Seite adjazent war. Zu beachten ist aber, daß auch bei diesem Ansatz die Ecken des umliegenden Graphen nur über Sorten angesprochen werden.

7.1.2 Nagl's Ansatz

In den meisten Ansätzen für Graphgrammatiken werden bei der Einbettungsfunktion nur Ecken des umliegenden Graphen, die sogenannten *Außenecken*, berücksichtigt, die vor einer Anwendung adjazent zu Ecken des ersetzten Graphen waren. Der Ansatz von Nagl ist in diesem Punkt allgemeiner. Er berücksichtigt auch die Ecken des umliegenden Graphen, die über einen durch Sorten spezifizierten Pfad von einem der normalerweise verwendeten Außenecken aus erreichbar sind. Damit können u.U. Kanten zu *allen* Ecken des umliegenden Graphen in einem Anwendungsschritt gezogen werden.

Hier wird nur beispielhaft ein eingeschränkter Ansatz der *Tiefe 1* betrachtet werden. Die Tiefe 1 besagt, daß *nur* die normalen Außenecken bei der Einbettung berücksichtigt werden. Da bei diesem Ansatz die Außenecken und die gelöschten Ecken über ihre Sorten identifiziert werden, entspricht die Einbettung exakt der allgemeinen Einbettung.

Nagl's Ansatz könnte aber bei bestimmten Restriktionen auch zu den konstruktiven Ansätzen gerechnet werden. Bei einer Produktionsanwendung wird zwar das Vorkommen der linken Seite komplett gelöscht, doch können bei seinem Ansatz auch bewahrende Teile spezifiziert werden. Diese werden dann mit der rechten Seite wieder eingefügt. Die komplette Löschung erlaubt es aber, u.U. die Orientierung von Kanten bzw. die Beschriftung von Kanten und Ecken zu ändern, ohne die Struktur selbst zu modifizieren. Aber auch bei Nagl's Ansatz ist zu beachten, daß bei der Einbettung die Ecken nur über Sorten angesprochen werden bzw. nur mit Sorten identifiziert werden.

Bei einer Produktionsanwendung auf einen Graphen G ist ferner zu beachten: Das Vorkommen der linken Seite in G bzw. der zu der linken Seite isomorphe Teilgraph von G muß ein Untergraph[3] von G sein.

[3]Untergraphen, wie in diesem Buch definiert, werden oft auch als induzierte Teilgraphen bezeichnet.

7.1.3 Web

Eine der ältesten Ansätze für Graphgrammatiken sind die *Web* Graphgrammatiken. In diesem Ansatz werden ungerichtete und Ecken beschriftete Graphen, die sogenannten *Webs*, verwendet. Als *Unterweb*, ein Pendant zu Teilgraphen, werden Untergraphen bezeichnet. In diesem Punkt entsprechen die Web Graphgrammatiken dem Ansatz von Nagl.

Analog zu den NCE Graphgrammatiken wird bei Webs die Einbettung vorgenommen. Hier muß die Verknüpfungsrelation C auch statt über Sorten teilweise über Ecken definiert werden, etwa der Form $C_{out} \subseteq V_L \times \Sigma_E \times \Sigma_V \times \Sigma_E \times V_R$, wobei V_R bzw. V_L jeweils die Ecken der rechten bzw. linken Seite einer Produktion sind. Der modifizierten Verknüpfungsrelation C entspricht die Funktion $\phi \subseteq V_L \times V_R \times \Sigma_V$.

Web Graphgrammatiken sind also eine Art Kombination der NCE Graphgrammatiken und des Ansatzes von Nagl's Graphgrammatiken der Tiefe 1. Von dem Ersten wurde die Einbettungsfunktion *übernommen*[4] und von dem Zweiten die Restriktion des induzierten Untergraphen für das Vorkommen einer linken Seite.

7.2 Konstruktive Ansätze

Im Gegensatz zu dem vorangegangen Abschnitt, werden in diesem Teil nur Graphgrammatiken behandelt, für die der *bewahrende* Teil und die Klebepunkte nicht leer sind. Es gilt also $gl(R) \neq \lambda \neq gl(L) \subseteq pr(L)$.

Des weiteren wird angenommen, daß $con(L) \subseteq pr(L)$ gilt. Damit werden bei einem Ableitungsschritt keine Kanten zu dem umliegenden Graphen entfernt; die Verknüpfungsrelation C ist dann überflüssig und wird daher als leer angenommen. Die Einbettung wird dann ganz durch die Kleberelation *apt* realisiert.

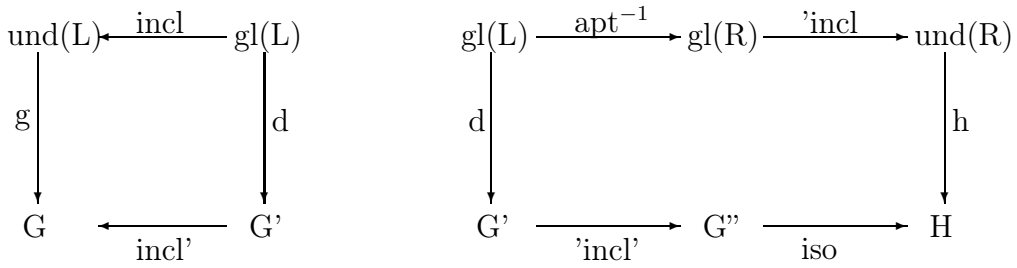
7.2.1 Berliner Ansatz

Der Berliner Ansatz konzentriert sich auf algebraische Graphgrammatiken, die eine kategorientheoretische Charakterisierung der Ableitung vornehmen. Bei diesem Ansatz sind die Produktionen von der Form $p = (L, R, apt, \emptyset)$, wobei $V_{con(L)} = V_{pr(L)}$ und $gl(L) = pr(L)$.

Sei für eine Anwendung einer Produktion p auf einen Graphen G $g : und(L) \to G$ der Graphmorphismus für das Vorkommen der linken Seite von

[4] *übernommen* ist hier nicht ganz richtig, da die Webs eigentlich etwas älter sind.

p in G und G‘ der Restgraph, d.h. G ohne $g(und(L) \setminus pr(L))$. Seien weiter $inc : gl(L) \to und(L)$ und $incl$‘ : G‘ $\to G$ Inklusionen[5]. Die Einschränkung von g auf den Klebepunkt $gl(L)$ $d : gl(L) \to G$‘ ist dann durch $g \circ incl = incl$‘ $\circ\, d$ wohldefiniert[6]. Das zugehörige kommutative Diagramm ist in der folgenden Abbildung auf der linken Seite dargestellt.



Analog wird nun für die rechte Seite der Produktion ein ähnliches kommutatives Diagramm definiert, dargestellt auf der rechten Seite der letzten Abbildung. Sei dazu nun d wie eben, G“ das Resultat der Einbettung der rechten Seite von p in G‘ und H ein Graph. Sei weiter apt^{-1} der zu apt inverse Isomorphismus, iso ein Graphisomorphismus und ‘$incl$ bzw. ‘$incl$‘ analog zu $incl$ Inklusionen. Sei schließlich h definiert durch $h(x) = iso(‘incl‘(d(y)))$, falls $‘incl(apt^{-1}(y)) = x$ und sonst $h(x) = iso(x)$.

In der Terminologie der Kategorientheorie wird das rechte Diagramm aus der Abbildung *Pushout* genannt. Gilt für g, daß die identifizierten Elemente, d.h. Kanten und Ecken, zu dem Klebepunkt gehören, ist die linke Seite in besagter Abbildung ebenfalls ein Pushout.

7.2.2 Edge Replacement

Bei einer Produktion $p = (L, R, apt, \emptyset)$ einer Edge Replacement Graphgrammatik ist die linke Seite L so, daß $con(L) = pr(L) = gl(L)$ Graphen sind, und daß $und(L)$ ein sogenannter *handle* ist, d.h. aus zwei adjazenten Ecken besteht. $con(L)$ und damit auch $pr(L)$ und $gl(L)$ bestehen aus diesen beiden Ecken. Die linke Seite L ist bei diesen Voraussetzungen dann ganz durch die Sorte der Kante spezifiziert. Die rechte Seite R ist auch ein Graph mit zwei Ecken, wobei $und(R) = gl(R)$.

Bei der Anwendung einer Produktion wird von dem zu L isomorphen Teilgraphen die Kante entfernt und gemäß apt bzw. R die beiden beteiligten Ecken

[5]Einschluß

[6]$\circ$ bezeichnet hier die Konkatenation von Funktionen, d.h. $f \circ g(a) = f(g(a))$.

verklebt. Ist R diskret, wird keine neue Kante eingefügt. Ist R ein handle wird eine neue Kante eingefügt, die gegenüber der gelöschten Kante u.U. umorientiert oder umbeschriftet ist. Es ist zu beachten, daß keine Ecken modifiziert werden.

7.2.3 Hypergraph Systeme

Hypergraph Systeme gehören ebenfalls in die Klasse der algebraischen Graphgrammatiken. Sie verwenden aber im Gegensatz zu dem Berliner Ansatz Hypergraphen, d.h. Graphen, deren Kanten mehr als eine Quell- oder Zielecke haben können.

Zur Modellierung dieser Systeme mit dem hier verwendeten Ansatz sei $pr(L) = und(L)$ für jede Produktion $p = (L, R, apt, \emptyset)$. Darüber hinaus kann $V_{con(L)} = V_{und(L)}$ sein; bei einer Produktionsanwendung auf einen Graphen G ist der Restgraph G‘ dann ebenfalls durch G gegeben, d.h. G‘ $= G$. Eine Ableitung resultiert damit in der *Verklebung* von G bzw. G‘ mit der rechten Seite bzw. $und(R)$. Diese Situation entspricht genau dem rechten Diagramm in der Abbildung des Berliner Ansatzes.

Eine Menge von Produktionen der beschriebenen Weise kann auch durch einen Graphen G und eine Familie von Teilgraphen $G_1, G_2, \ldots G_n$ von G beschrieben werden. Für $i \neq j, 1 \leq i, j \leq n$ mit $G_i \cap G_j \neq \lambda$, d.h. G_i und G_j haben einen gemeinsamen Teilgraphen, wird eine Produktion $p = (L, R, apt, \emptyset)$ beschrieben durch

- $und(L_{ij}) = pr(L_{ij}) = G_i$,
- $V_{con(L_{ij})} = V_{G_i}$ und damit insbesondere $V_{con(L_{ij})} = V_{und(L_{ij})}$,
- $und(R_{ij}) = G_j$,
- $gl(L_{ij}) = gl(R_{ij}) = G_i \cap G_j$ und
- $apt_{ij} = id_{G_i \cap G_j}$, d.h. die Identität auf $G_i \cap G_j$.

Ein Hypergraph System kann nun wie folgt beschrieben werden: Sei G ein unbeschrifteter Graph und seien die Teilgraphen $G_1, \ldots G_n$ paarweise verschiedene Untergraphen von G. Dann läßt sich jeder Untergraph G_i durch die Menge seiner Ecken beschreiben, die als Hyperkante des Graphen G interpretiert werden kann. Daher kann G mit den Untergraphen $G_1, \ldots G_n$, wobei G_1 der initiale Graph sei, zur Beschreibung eines Hypergraph System verwendet werden.

Diese Graphen G und $G_i, i \in \{1, \ldots n\}$ beschreiben nun eine Graphgrammatik, wobei die Produktionen von eben beschriebener Gestalt sind und G_1 das Axiom, d.h. der Startgraph ist. Des weiteren sind $\Sigma_V = \{*\} = \Sigma_E$. Für

eine Ableitungssequenz $G \Rightarrow^{p_{ij}} G' \Rightarrow^{p_{kl}} G''$ muß gelten, daß $j = k$, d.h. $und(R_{ij}) = G_j = G_k = und(L_{kl})$, und darüber hinaus muß auch das Vorkommen der linken Seite von p_{kl} dem Vorkommen der rechten Seite von p_{ij} entsprechen.

Index