Das Knoten-überdeckungsproblem

Eine Fallstudie zur Didaktik NP-schwerer Probleme

Teil 2

von Rolf Niedermeier, Jörg Vogel, Michael Fothe und Mirko König

In diesem Beitrag wird das Thema „NP-schwere Probleme“ für die Schule aus didaktisch-methodischer Sicht aufbereitet. Stellvertretend geschieht dies am Knotenüberdeckungsproblem, das erhebliche praktische Relevanz besitzt. Im ersten Teil (siehe LOG IN, Heft 146/147, S. 53 ff.) wurden Beispiele sowie Modellierungsaspekte zur Knotenüberdeckung in Graphen aufgezeigt. Ausgehend von unterschiedlichen Modellierungen wurden verschiedene Lösungsverfahren erläutert und bewertet. Darüber hinaus wurde die mit NP-schweren Problemen verbundene, anscheinend unvermeidliche „kombinatorische Explosion“ thematisiert.

Ausgehend vom konkreten Problem werden im Folgenden Problemverwandtschaft, Determinismus und Nichtdeterminismus intuitiv erklärt. Dies mündet in Erläuterungen zur P=NP-Frage, einem der interessantesten Probleme der aktuellen Forschung in der Informatik.

Das Knoten-überdeckungsproblem als NP-vollständiges Problem

Die im 1. Teil dieses Beitrags beschriebenen Beobachtungen sind typisch für sogenannte *NP-schwere Probleme*. Das Knotenüberdeckungsproblem ist ein solches Problem. Darüber hinaus trifft auch ein weiteres Phänomen zu, das im Folgenden dargestellt werden soll. Eine typische Eigenschaft klassischer Algorithmen ist nämlich ihre Determiniertheit: Jeder Schritt legt eindeutig den Folgeschritt fest.

Wenn ein binärer Baum vollständig durchsucht werden soll, muss zunächst eine Suchstrategie festgelegt werden, z. B. Tiefensuche. Nach dieser Strategie wird der gesamte Baum in eindeutig bestimmter Weise durchlaufen. Dies erfordert für einen Baum der Tiefe n einen Aufwand, der proportional zu 2^n ist, d. h. exponentiell wachsenden Zeitaufwand.

Nichtdeterminismus erlaubt eine starke Reduzierung dieses Aufwands.

Wir beschreiben dieses Phänomen zunächst als eine Schatzsuche. Das Ziel beim Durchwandern aller Blätter eines vollständigen Suchbaums war im Abschnitt „Beispiele zur Knotenüberdeckung in Graphen“ des 1. Teils dieses Beitrags (siehe LOG IN 146/147, S. 53–55) das Finden einer minimalen Knotenüberdeckungsmenge. Dies sei unser Schatz, der in einem Blatt des Suchbaums versteckt ist. Da über den Ort des Schatzes nichts bekannt ist, muss angenommen werden, dass dieser Schatz gleichverteilt über alle Blätter versteckt ist. Damit ist für einen Einzelsucher keine Suchstrategie bekannt, die nicht mindestens exponentiellen Aufwand erwarten lässt.

Ganz anders ist es bei einer Suche, an der 2^n Personen beteiligt sind. Startpunkt der Suche ist die Wurzel des Baums, und man sucht in Richtung der Blätter. An jeder Verzweigung teilt sich die Gruppe immer wieder, bis die Blätter erreicht sind. Danach laufen alle zur Wurzel zurück. Die Suche ist erfolgreich, wenn eine Person den Schatz gefunden hat und ihn zur Wurzel mitbringt.

Der Effekt einer so organisierten Suche ist gewaltig: Der Aufwand der zuletzt beschriebenen nichtdeterministischen Suche ist nur noch proportional zu n. Der Preis hierfür sind die 2^n Sucher.

Dieses Phänomen lässt sich auch durch „Raten“ beschreiben: An jeder Verzweigung „rät“ man den richtigen Weg, und die Suche wird erfolgreich, wenn es einen Weg zum Schatz gibt. Die Übertragung auf das Knotenüberdeckungsproblem bedeutet:

In einer ersten nichtdeterministischen „Ratephase“ wird eine Teilmenge U der Knotenmenge geraten. Der Aufwand hierfür ist proportional zu n, er ist also polynomial in n beschränkt; in einer zweiten (deterministischen) „Prüfphase“ wird verifiziert, ob U tatsächlich eine Menge ist, die die Knotenüberdeckung leistet: Wie bereits erwähnt ist der Aufwand hierfür ebenfalls polynomial beschränkt.

Zusammenfassend können wir feststellen: Das Knotenüberdeckungsproblem erlaubt eine präzise Lösung mit polynomialem Aufwand, falls Nichtdeterminismus zugelassen ist. Dieses Phänomen ist typisch für sogenannte *NP-Probleme*. Das Knotenüberdeckungsproblem ist ein solches NP-Problem.

NP-Probleme, die zugleich NP-schwer sind, heißen *NP-vollständige* Probleme.

Das Knotenüberdeckungsproblem ist also ein NP-vollständiges Problem. Im folgenden Abschnitt sollen diese Begriffe näher erläutert werden.

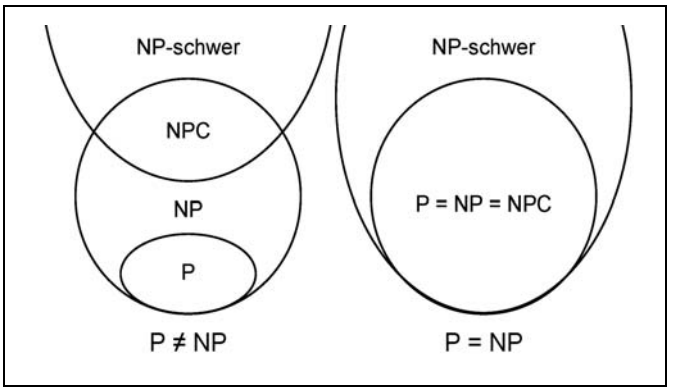

Bild 1: Zusammenhang zwischen den Komplexitätsklassen P, NP, NPC und NP-schwer.

NP-vollständige Mengen und das P-NP-Problem

Das *P-NP-Problem* (oder auch $P \stackrel{?}{=} NP$ bzw. *P versus NP* genannt) ist ein offenes Problem der theoretischen Informatik. Wie im 1. Teil dieses Beitrags schon erwähnt (siehe LOG IN 146/147, S.53), wurde die P=NP-Frage erstmals 1971 von Stephen Cook (USA bzw. Kanada) und im gleichen Jahr von Leonid Levin (damalige UdSSR) gestellt. Es wurde vom *Clay Mathematics Institute* (Cambidge, MA, USA) in die Liste der sogenannten Millennium-Probleme aufgenommen, für deren Lösung jeweils eine Million US-Dollar ausgelobt wurde. Das P-NP-Problem beinhaltet die Frage nach der Inklusionsbeziehung zwischen den Komplexitätsklassen P und NP. Wie nachfolgend erläutert, gilt $P \subseteq NP$, wobei jedoch unklar ist, ob $P = NP$ oder $P \subset NP$ gilt.

NP bezeichnet die Klasse aller Probleme, bei denen es mit einem deterministischen Algorithmus in Polynomialzeit möglich ist zu verifizieren, ob eine geratene Lösung tatsächlich eine Lösung darstellt. Der Begriff des Algorithmus kann hierbei durch das Konzept der Turing-Maschine formalisiert werden. Entsprechende Turing-Maschinen arbeiten üblicherweise in zwei Phasen: In der ersten Phase wird nichtdeterministisch eine Lösung geraten, in der zweiten Phase wird diese geratene Lösung deterministisch verifiziert.

Es gibt eine zweite äquivalente Definition von NP als die Menge aller Entscheidungsprobleme, die von nichtdeterministischen Turing-Maschinen in Polynomialzeit akzeptiert werden. NP enthält also alle Probleme, die von nichtdeterministischen Algorithmen mit polynomialem Zeitaufwand gelöst werden können.

Die Komplexitätsklasse P dagegen enthält alle Probleme, die von klassischen Algorithmen mit polynomialem Zeitaufwand gelöst werden können. Formal ausgedrückt umfasst die Klasse P alle Entscheidungsprobleme, die von deterministischen Turing-Maschinen in Polynomialzeit entschieden werden können. Aus diesen beiden Definitionen geht hervor, dass P eine Teilmenge von NP ist (siehe auch Bild 1), denn deterministische Turing-Maschinen sind auch nichtdeterministische Turing-Maschinen, denen die Zufälligkeit, also das Raten, fehlt.

Intuitiv ist ein Problem NP-vollständig (NPC), wenn es erstens zur Klasse NP gehört, d.h. wenn es ein NP-Problem ist, und wenn es zweitens die Eigenschaft hat, dass sich die Schwierigkeit jedes Problems aus NP in das gegebene übersetzen lässt, d.h. wenn jedes NP-Problem auf das gegebene Problem polynomial reduzierbar ist. Folglich lassen sich auch alle NP-vollständigen Probleme ineinander übersetzen, und jedes einzelne trägt die Komplexität der ganzen Klasse in sich. Probleme, die diese zweite Eigenschaft besitzen, heißen *NP-schwer*. NP-schwere Probleme sind also solche, die die Komplexität aller NP-Probleme in sich tragen.

Alle NP-vollständigen Probleme haben zwei Gemeinsamkeiten:

- ▷ Zum einen benötigen alle bekannten deterministischen Algorithmen, die eine exakte Lösung garantieren, Exponentialzeit, d.h. exponentiellen Rechenzeitaufwand zum Finden der korrekten Antwort.
- ▷ Zum anderen hätte man für alle NP-vollständigen Probleme Lösungsalgorithmen in Polynomialzeit, falls es gelänge, für nur ein einziges Problem einen derartigen deterministischen Algorithmus zu finden. Dies hätte die Gleichheit P = NP zur Folge.

Die Lösung des P-NP-Problems ist daher von großer praktischer Bedeutung. Denn sehr viele in der Praxis relevante Probleme sind NP-vollständig. Der Beweis von P = NP würde bedeuten, dass für Probleme der bisherigen Klasse NP tatsächlich Algorithmen existieren, deren jeweilige Lösung in wesentlich geringerer Zeit und somit Polynomialzeit erzeugt werden kann. Diese Gleichheit ist zwar prinzipiell möglich, aber von vielen Experten wird sie eher als unwahrscheinlich angesehen.

Eine Umfrage zur P=NP-Frage im Jahre 2002 unter 100 namhaften Wissenschaftlern aus der theoretischen Informatik lieferte folgende Ergebnisse (vgl. Gasarch, 2002): Lediglich fünf der Befragten äußerten die Auffassung, dass diese Frage bis zum Jahr 2009 beantwortet werden könne. Immerhin 35 waren der Ansicht, dass das Problem bis zum Jahr 2049 gelöst sein werde. Sieben sagten, es werde in den Jahren zwischen 2100 und 2110 gelöst werden, weitere fünf meinten dagegen, dies sei erst in den Jahren zwischen 2200 und 2300 der Fall. Immerhin gaben fünf Befragte an, dass es niemals gelöst werden würde.

Die Befragung zeigte weiter: 61 der Befragten hatten die Überzeugung, die korrekte Antwort sei $P \subset NP$; neun gaben der Überzeugung $P = NP$ Ausdruck. Und es wurde eigens erwähnt, dass es sich bei diesen neun Personen um sehr „respectable members of the community" handelte, denen klar war, dass sie eine Minderheitenmeinung äußerten.

Der Umstand, dass 22 der Befragten das Risiko scheuten, sich festzulegen, macht deutlich, dass hinter dieser Frage eine große Unbestimmtheit steckt.

Lösungsstrategien

Im Allgemeinen ist das Knotenüberdeckungsproblem NP-schwer, und so ist nicht zu hoffen, einen im Allgemeinen effizienten Lösungsalgorithmus hierfür zu finden. Dennoch bedarf dieses Problem der algorithmischen Behandlung und nachfolgend werden einige Lösungsansätze erläutert.

Ein Lösungsansatz besteht darin, die Suche nach einer optimalen Lösung aufzugeben und sich stattdessen mit einer Näherungslösung zufrieden zu geben. Solche Verfahren kennt man als (Polynomialzeit-)*Approximationsalgorithmen*. Für das Knotenüberdeckungsproblem ist schnell eine Lösung zu finden, die garantiert höchstens doppelt so viele Knoten enthält wie eine optimale.

Die zugrunde liegende Idee ist einfach: Gemäß Definition des Problems muss jede Kante durch einen ihrer beiden Endknoten abgedeckt werden. Das Problem ist, dass im Allgemeinen nicht schnell zu sehen ist, welche Wahl die günstigere ist. Der Approximationsalgorithmus wählt daher schlicht beide Endknoten der Kante. Mindestens einer der beiden Endknoten ist auf jeden Fall in einer optimalen Lösung. Wählt man nun immer wieder eine beliebige Kante, löscht ihre beiden Endknoten zusammen mit allen anliegenden Kanten, bis keine Kanten im Graphen mehr übrig sind, so endet man aufgrund obiger Beobachtung mit einer Lösung, die höchstens doppelt so groß ist wie eine optimale – man spricht von einer *Faktor-2-Approximation*. Da der Algorithmus immer die erstbeste nichtabgedeckte Kante nehmen darf, ist leicht einzusehen, dass er in linearer Laufzeit arbeitet, d. h. extrem effizient ist. Einziger, aber entscheidender Wermutstropfen ist die Nichtoptimalität der Lösungsgröße.

Eine weitere effiziente, aber im Allgemeinen auch nicht die optimale Lösung liefernde Strategie beruht auf folgender intuitiver Idee: Je mehr Kanten an einem Knoten anliegen, desto mehr Kanten werden bei Wahl dieses Knotens in die Knotenüberdeckung auf einmal abgedeckt. Daher liegt es nahe, jeweils denjenigen Knoten zu wählen, der die aktuell meisten Kanten abdeckt, ihn und alle anliegenden Kanten zu löschen, und zu iterieren, bis wiederum alle Graphkanten abgedeckt (d. h. gelöscht) wurden. Überraschenderweise führt diese Strategie schlimmstenfalls zu einer schlechteren Approximation als die im vorigen Abschnitt beschriebene Strategie. Genauer lässt sich (nicht ganz einfach) ein Graph mit n Knoten so konstruieren, dass für ihn die derart konstruierte Lösungsmenge die $(\ln n)$-fache Knotenanzahl im Vergleich mit einer optimalen Lösung besitzt. Jedoch ist zu vermuten, dass dieser Effekt eher nur in sehr speziellen Fällen auftritt. So erklärt sich, dass diese Strategie trotz ihres deutlich schlechteren *worst-case*-Verhaltens hinsichtlich Approximationsgüte in praktischen Anwendungsfällen meist bessere Resultate liefert als die Faktor-2-Approximation.

Es lässt sich festhalten, dass das Knotenüberdeckungsproblem eine Vielzahl möglicher Lösungsstrategien besitzt, die entweder näherungsweise Lösungen in polynomialer Laufzeit oder optimale Lösungen in exponentieller Laufzeit liefern. Sämtliche Verfahren sind leicht zu verstehen und z. T. sogar spielerisch zu erfassen. Somit können hier auf einfache Weise zentrale Themen der Algorithmik – approximative, exakte und heuristische Algorithmen für NP-vollständige Probleme – anschaulich vermittelt werden.

Beispiele für den Unterricht

von Rüdeger Baumann

Ein großer Vorzug der Informatik (in der Schule) besteht darin, dass Fragen der theoretischen Informatik, insbesondere der Komplexitätstheorie, nicht „rein theoretisch", d. h. praxisfern abgehandelt werden müssen, sondern dass sie sich mittels Anwendungsproblemen motivieren lassen, und dass die mithilfe theoretischer (mathematisch-logischer) Argumentation gewonnenen Aussagen anhand von Programmen quasi experimentell überprüft (wenn auch nicht bewiesen) werden können. Die in den vorstehenden Abschnitten zum Knotenüberdeckungsproblem angegebenen Praxisbeispiele und theoretischen Überlegungen werden somit erst dann unterrichtlich fruchtbar, wenn nicht beim mathematischen Modellieren Halt gemacht, sondern bis zum Implementieren und anschließenden Experimentieren mit dem Programm vorangeschritten wird. Mit Recht heißt es in den Bildungsstandards: „Beim informatischen Modellieren ist die Implementierung unverzichtbar" (Arbeitskreis „Bildungsstandards" der Gesellschaft für Informatik, 2007, S. 52). Mit der Entwicklung von Datenstrukturen (hier: für Graphen) und Algorithmen (hier: zur Konstruktion von Knotenüberdeckungen) beginnt damit erst eigentlich die Tätigkeit des Informatikers – nachdem zuvor der Fachwissenschaftler des jeweiligen Anwendungsgebiets tätig war.

Ein naiver Algorithmus

Zur klärenden Wiederholung seien einige Definitionen und Bezeichnungen vorangestellt, die im Folgenden verwendet werden.

Ein *ungerichteter Graph* besteht aus einer endlichen Menge von *Knoten* (auch *Punkte* oder *Ecken* genannt), die durch Linien (die sogenannten *Kanten*) miteinander verbunden sind; wir schreiben im Folgenden $G = (E, K)$, wobei E die Menge der Ecken und K die der

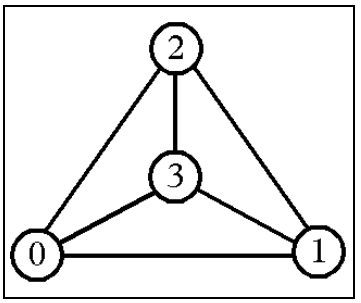

Bild 2: Tetraedergraph.

Kanten ist. Dienen Graphen zur Modellierung von Anwendungssituationen, spricht man gern von Knoten; stellt man sie geometrisch dar, eher von Ecken oder Punkten. Eine andere Schreibweise (siehe Teil 1 in LOG IN 146/147, S. 55) ist auch G = (V, E), wobei V die Knotenmenge (*set of vertices*) und E die Kantenmenge (*set of edges*) wäre. Sind p, q zwei Knoten von G, so bezeichnen wir ihre Verbindungskante mit k = pq und sagen, p und q seien *adjazent* (benachbart) und sie seien mit k *inzident* (Endpunkte von k).

Eine Teilmenge U von E heißt *Knotenüberdeckung* von G (oder: *Eckenüberdeckung*; engl.: *vertex cover*), wenn es zu jeder Kante von G einen mit ihr inzidenten Knoten in U gibt. Im Bild 2 bilden die Knoten (bzw. Ecken) der Tetraeder-Grundfläche, also die Menge {0, 1, 3}, eine Knotenüberdeckung. Aus Symmetriegründen gilt dies auch für die Knoten der drei Seitenflächen des Tetraeders. Diese Überdeckungen sind zudem bezüglich der Anzahl minimal, d. h. keine Überdeckung des Tetraeders kann mit weniger als drei Knoten auskommen.

Allgemein heißt eine Knotenüberdeckung *anzahlminimal*, wenn es keine Knotenüberdeckung mit weniger Elementen gibt; diese Anzahl heißt kurz *Überdeckungszahl*.

Das im 1. Teil dieses Beitrags vorgestellte *Experimentproblem* (siehe LOG IN 146/147, S. 54) lässt sich als *Problem der unabhängigen Mengen* auffassen: Gesucht ist eine Menge wechselseitig einander nicht widersprechender Hypothesen. Eine Teilmenge der Knotenmenge E des Graphen G = (E, K) heißt *unabhängig* (oder: *stabil*; engl.: *independent*), wenn ihre Knoten paarweise nicht-adjazent sind. Die Knotenmenge U ist genau dann eine Überdeckung, wenn ihr Komplement E\U unabhängig ist (siehe LOG IN 146/147, S. 56).

Aufgabe 1: Finden Sie zum Oktaeder (Bild 3a) und zum Hexaeder (Bild 3b) eine anzahlminimale Knotenüberdeckung (*Hilfe:* die Überdeckungszahl ist in beiden Fällen 4).

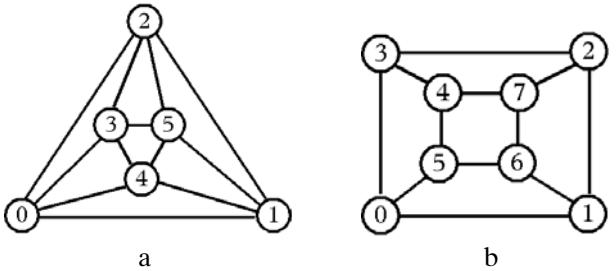

Bild 3: Oktaedergraph (a), Hexaedergraph (b).

Aufgabe 2: Geben Sie die unabhängigen Mengen zum Oktaeder (Bild 3a) und zum Hexaeder (Bild 3b) an.

Um minimale Knotenüberdeckungen des Graphen G = (E, K) zu finden, suchen wir nach maximalen unabhängigen Mengen. Der *naive Algorithmus* inspiziert dazu systematisch alle Teilmengen von E. Wir erzeugen diese rekursiv, indem wir eine (zu Beginn leere) Menge *Unabh* schrittweise dadurch erweitern, dass wir ihr alle nicht-adjazenten Knoten einverleiben. Letztere entnehmen wir der Menge *Kand* (der „Erweiterungskandidaten"), die zu Beginn aus der gesamten Knotenmenge E besteht. Der Algorithmus lautet:

SucheUnabh (U, Kand)

```
SOLANGE Kand ≠ leere Menge WIEDERHOLE {
   Entferne p aus Kand und füge p in U ein
   KandNeu = Menge der zu p nicht-adjazenten Knoten
         von Kand
   WENN KandNeu = leere Menge DANN Ausgabe: U
   SucheUnabh(U, KandNeu) // rekursiver Aufruf
   Entferne p aus U
   } // Ende-wiederhole
```

Um das Verfahren in JAVA zu implementieren, wird den Schülerinnen und Schülern eine Klasse Graph zur Verfügung gestellt und dessen „Gebrauchsanleitung" als Spezifikation des entsprechenden *abstrakten Datentyps* (ADT) angegeben. Unter einem ADT versteht man einen Datentyp (d. h. eine Menge von Werten und von Operationen auf diesen Werten), der „abstrakt", d. h. implementationsunabhängig gegeben ist. Dies macht es möglich, Exemplare des Datentyps zu verwenden, ohne deren Implementation zu kennen oder gar selbst entwickeln zu müssen.

Spezifikation des ADT Graph

```
Attribute
   AnzahlKnoten: Ganzzahl
   AnzahlKanten: Ganzzahl
   IstGerichtet: Wahrheitswert
Konstruktoraufruf
   Graph(10, falsch)
         // Beispiel: 10 Knoten, ungerichteter Graph
Methoden
   EinfügeKante(p, q)
   EntferneKante(p, q)
   IstAdjazent(p, q): Wahrheitswert
         // p und q sind benachbart
   Adjazenz(p): Knotenliste
         // Liste der zu p benachbarten Knoten
```

Die n Knoten eines Graphen repräsentieren wir am einfachsten durch die Menge {0, 1, …, n – 1}; in JAVA entspricht dem der einfache Datentyp int bzw. der Ob-

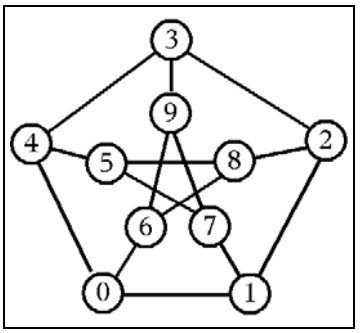

Bild 4: Petersen-Graph.

jekttyp Integer. Die Kanten pq werden durch eine lokale Klasse wie folgt realisiert:

```
class Kante {
  int p, q;
  Kante (int p, int q) {this.p = p; this.q = q;}
  } // Ende Kante
```

Ein einzelner Graph (z. B. der Petersen-Graph, nach dem dänischen Mathematiker Julius Petersen benannt, siehe Bild 4) wird als Klasse, die die Klasse Graph erweitert, wie folgt dargestellt:

```
public class Petersen extends Graph {

  public Petersen() {
    super(10, false);
    setzName("PETERSEN-GRAPH");

    einfüge(new Kante(0, 1));
    einfüge(new Kante(0, 4));
    einfüge(new Kante(0, 6));

  } // Ende Petersen
```

Aufgabe 3: Der naive Algorithmus zur Suche maximaler unabhängiger Mengen (bzw. minimaler Knotenüberdeckungen) soll implementiert und u. a. am Petersen-Graphen getestet werden.

Wir entwickeln eine Klasse Naivdecke, der wir ein Exemplar des ADT Graph übergeben. Ihr Konstruktor sieht wie folgt aus:

```
public Naivdecke (Graph g) {
  graph = g;
  ergebnis = "";
  maximum = 0;
  } // Ende Konstruktor

public String Ergebnis() {
  sucheUnabh(new ArrayList(),
             Kopie(graph.Knotenliste()));
  return ergebnis;
  } // Ende Ergebnis
```

Die Prozedur sucheUnabh() implementiert den obigen gleichnamigen Algorithmus und bildet zugleich das Komplement von Unabh, sodass als Ergebnis eine minimale Knotenüberdeckung geliefert wird:

```
private void sucheUnabh (ArrayList Unabh,
             ArrayList Kand) {
  while (! Kand.isEmpty()) {
    Integer p = (Integer) Kand.remove(0);
    Unabh.add(p);
    ArrayList KandNeu =
      Schnitt(Kand,
              graph.NichtAdjazenz(p.intValue()));
    if (KandNeu.isEmpty()) {
      if (maximum <= Unabh.size()) {
        maximum = Unabh.size();
        ArrayList Decke = graph.Komplement(Unabh);
        ergebnis += graph.Knotenausgabe(Decke)
                 + "\n";
        } // Ende if
      } // Ende if
    sucheUnabh(Unabh, KandNeu);
    Unabh.remove(p);
    } // Ende while
  } // Ende sucheUnabh
```

Das Programm Graphtest erzeugt zunächst einen neuen Graphen, stellt ihn auf dem Bildschirm dar, übergibt ihn an die Klasse Naivdecke und ruft deren Funktion Ergebnis() auf:

```
public class Graphtest {

  public static void main (String[] xy) {
    Graph g = new Petersen();
    System.out.print(g.Gesamtausgabe());
    System.out.println(new
                       Naivdecke(g).Ergebnis());
    } // Ende main

  } // Ende Graphtest
```

Das Programm liefert fünf minimale Überdeckungen (siehe Bild 5), die die fünfzählige Symmetrie des Petersen-Graphen widerspiegeln.

Aufgabe 4: Das Programm ist so abzuändern, dass es nach Eingabe einer positiven ganzen Zahl k die Frage beantwortet, ob eine Knotenüberdeckung mit höchstens k Elementen existiert oder nicht. (Es handelt sich jetzt nicht um ein Optimierungs-, sondern um ein Entscheidungsproblem; siehe auch LOG IN 146/147, S. 55).

```
PETERSEN-GRAPH

Kn: Nachbarn
00: 01 04 06
01: 00 02 07
02: 01 03 08
03: 02 04 09
04: 00 03 05
05: 04 07 08
06: 00 08 09
07: 01 05 09
08: 02 05 06
09: 03 06 07

10 Knoten, 15 Kanten

01 03 04 06 07 08
01 02 04 05 06 09
00 02 04 07 08 09
00 02 03 05 06 07
00 01 03 05 08 09
```

Bild 5: Minimale Überdeckungen des Petersen-Graphen.

Experimente mit Zufallsgraphen

Um die relative (Un-)Brauchbarkeit des naiven Algorithmus zu demonstrieren und für die Schülerinnen und Schüler erfahrbar zu machen, bedienen wir uns sogenannter *Zufallsgraphen*. Dieser Begriff kann in unterschiedlicher Weise konkretisiert werden; wir entscheiden uns für folgende Version (siehe Diestel, [3]2006, S. 250):

Zufallsgraph

Für alle Knotenpaare k der Menge E = {0, 1, …, n – 1} wird mit Wahrscheinlichkeit p entschieden, ob k eine Kante des Graphen sein soll oder nicht; dabei ist p (die sogenannte *Kantendichte*) ein fester Wert zwischen 0 und 1.

Diese Spezifikation wird in folgender Klasse realisiert:

```
public class Zufallsgraph extends Graph {

  public Zufallsgraph (int n, double p) {
    super (n, false);
    for (int i = 0; i < n - 1; i++)
      for (int j = i + 1; j < n; j++)
        if (Math.random() < p)
            einfüge(new Kante(i, j));
    } // Ende Konstruktor

  } // Ende Zufallsgraph
```

Aufgabe 5: Man schreibe ein auf einen solchen Zufallsgraphen aufbauendes Testprogramm und beurteile an Beispielen den mit der Knotenanzahl wachsenden Zeitbedarf.

Näherungsverfahren

Nunmehr sollen die im vorliegenden Teil des Beitrags skizzierten Näherungsverfahren in JAVA umgesetzt und getestet werden. Wir greifen uns irgendeine Kante k = pq heraus, merken uns ihr Endpunkte-Paar, d. h. die Knoten am jeweiligen Ende der Kante, und streichen sodann alle damit inzidenten Kanten. Mit den verbleibenden Kanten verfahren wir auf die gleiche Weise solange, bis alle aufgebraucht sind.

Paarungsalgorithmus

Eingabe: G = (E, K) // ungerichteter Graph
U = leere Menge
SOLANGE K nicht leer WIEDERHOLE {
  (*) wähle Kante pq beliebig aus K
  füge p und q zu U hinzu
  (**) lösche alle mit p oder q inzidenten Kanten aus K
  } // Ende-wiederhole
Ausgabe: U // Knotenüberdeckung

Am Graphen von Bild 6 sieht die Ausführung des Algorithmus wie folgt aus: Es wird zuerst Kante 0-2 gewählt und U = {0, 2} gesetzt; sodann werden die Kanten 0-3, 0-5, 2-1, 2-6, 2-9 aus K gestrichen. Die nächste Kante ist 1-4, U = {0, 2, 1, 4}; gestrichen werden: 1-5, 4-7, 4-8, 4-9. Nach Wahl von 3-6 und 5-8 ergibt sich U = {0, 2, 1, 4, 3, 6, 5, 8}. Es ist leicht zu sehen, dass die minimale Überdeckung aus den Knoten 2, 3, 4, 5 besteht. Die vom Algorithmus gelieferte Überdeckung enthält also doppelt so viele Knoten wie die minimale.

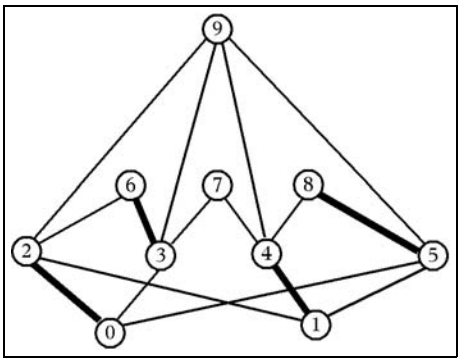

Bild 6: Die fett gezeichneten Kanten bilden eine Paarung des Graphen.

Der Name des Algorithmus rührt daher, dass er eine sogenannte *Paarung* (oder: *Zuordnung*; engl.: *matching*) des Graphen erzeugt. Darunter versteht man eine Teilmenge M von K mit der Eigenschaft, dass keine zwei Kanten von M einen Knoten gemeinsam haben. Die Paarung M ist *nicht-erweiterbar*, da es keine Kante k gibt derart, dass M nach Hinzunahme von k eine Paarung ist. Die Endpunkte einer nicht-erweiterbaren Paarung von G bilden eine Knotenüberdeckung U von G, denn gäbe es in K eine von U nicht abgedeckte Kante, so könnte man mit ihr die Paarung erweitern.

In JAVA lautet der Paarungsalgorithmus wie folgt:

```
private void erzeugeÜberdeckung (ArrayList K) {
  ArrayList U = new ArrayList();
  while (! K.isEmpty()) {
    Kante k = (Kante) K.get(0);
    U.add(new Integer(k.p));
    U.add(new Integer(k.q));
    lösche(graph.Inzidenz(k.p), K);
    lösche(graph.Inzidenz(k.q), K);
    } // Ende while
  ergebnis += graph.Knotenausgabe(U);
  } // Ende erzeugeÜberdeckung
```

Satz: Die vom Paarungsalgorithmus gelieferte Überdeckung besteht aus höchstens doppelt so vielen Knoten wie eine optimale (anzahlminimale) Überdeckung.

Beweis: Sei M die Menge der Kanten, die in Anweisung (*) des Algorithmus herausgegriffen werden. Keine zwei dieser Kanten haben einen Knoten gemeinsam, denn wurde pq gewählt, so löscht Anweisung (**) alle mit p und alle mit q inzidenten Kanten. Bei jedem Schleifendurchlauf kommen also jeweils zwei neue Knoten zu U hinzu, d. h. Anzahl(U) = 2 · Anzahl(M). Ist opt die Anzahl einer optimalen Knotenüberdeckung, so gilt Anzahl(M) ≤ opt, da jede Überdeckung von K, also auch die optimale, mindestens eine Kante von M abdeckt. Es folgt Anzahl(U) = 2 · Anzahl(M) ≤ 2 · opt. Man sagt, der *Approximationsfaktor* des Verfahrens habe den Wert 2.

Aufgabe 6: Bei einer „gierigen“ (engl.: *greedy*) Lösungsstrategie beginnt man mit einem Knoten höchsten Grades, d. h. mit der größten Anzahl adjazenter Knoten, streicht die Kanten und fährt auf diese Weise fort.

(a) Formulieren Sie dieses Verfahren in algorithmischer Schreibweise, testen Sie es an geeigneten Beispielen und implementieren Sie es.
(b) Finden Sie ein Beispiel, bei dem der gierige Algorithmus die optimale Lösung findet (siehe Bild 7).
(c) Zeigen Sie mittels numerischer Experimente, dass der gierige Algorithmus im Mittel besser ist als der Paarungsalgorithmus.
(d) Finden Sie ein Beispiel, für das der gierige Algorithmus ein schlechteres Ergebnis liefert als der Paarungsalgorithmus (siehe etwa Wanka, 2006, S. 64).

```
ZUFALLSGRAPH

Kn: Nachbarn
00: 01 02 08 09
01: 00 08
02: 00
03: 04 09
04: 03 07 09
05: 07 08
06: 07
07: 04 05 06
08: 00 01 05 09
09: 00 03 04 08

10 Knoten, 13 Kanten

01 02 04 07 08 09
00 04 07 08 09

Paarungsalgorithmus:
00 01 03 04 05 07 08 09

Gieriger Algorithmus:
00 08 09 04 07
```

Bild 7: Vergleich zweier Näherungsverfahren mittels Zufallsgraphen.

Das Cliquenproblem

Zweck des folgenden Unterrichtsabschnitts ist der Nachweis, dass man hinsichtlich exakter Verfahren nicht beim naiven Algorithmus – nämlich der vollständigen Enumeration (Aufzählung aller Teilmengen der Knotenmenge) – stehenbleiben muss, dass es vielmehr effizientere Verfahren gibt, die mit dem naiven allerdings den Nachteil gemeinsam haben, keine polynomiale Zeitkomplexität zu besitzen. Bereits oben haben wir gesehen, dass sich Knotenüberdeckungen mittels unabhängiger Mengen gewinnen lassen. Nunmehr schlagen wir einen Umweg über das Cliquenproblem ein und demonstrieren damit den Nutzen der Reduktion eines Problems auf ein anderes.

Das im 1. Teil des Beitrags (siehe LOG IN 146/147, S. 53) vorgestellte *Problem der harmonischen Geburtstagsfeier* („Partyproblem“) lässt sich auch als Cliquenproblem formulieren: Gesucht ist eine größtmögliche Menge der zur Teilnahme vorgesehenen Personen, von denen jede mit jeder harmoniert. Unter einer *Clique* versteht man einen vollständigen Teilgraphen, das heißt: Die Knoten einer Clique sind paarweise miteinander adjazent (jede ist zu jeder benachbart). Eine Clique des Graphen G heißt *inklusionsmaximal*, wenn sie in keiner anderen Clique von G enthalten ist; sie heißt *anzahlmaximal*, wenn es in G keine Clique mit mehr Knoten gibt. Das *Cliquenproblem* besteht nun darin, die anzahlmaximalen Cliquen in G zu finden. Beispielsweise ist der Tetraeder (Bild 2, siehe S. 84) selbst eine Clique, während der Oktaeder (Bild 3a, siehe S. 84) acht Dreiercliquen und der Hexaeder (Bild 3b, siehe S. 84) zwölf Zweiercliquen besitzt.

Das Cliquenproblem lässt sich auf das Knotenüberdeckungsproblem zurückführen, das heißt: Aus einer Lösung des letzteren lässt sich – auf effiziente Weise – eine Lösung des ersteren gewinnen. Im Geburtstagsfeier-Beispiel: Wenn wir wissen, welche Personen auszuschließen sind (weil sie mit mindestens einer Person nicht harmonieren), kennen wir auch die einzuladenden.

Um diesen Sachverhalt exakt zu formulieren, benötigen wir den Begriff des *Komplements* eines Graphen G = (E, K): Es besteht aus den gleichen Knoten wie G, wobei zwei Knoten in G* = (E, K*) genau dann adjazent sind, wenn sie es in G *nicht* sind (siehe LOG IN 146/147, S. 56). In JAVA:

```
public class Komplement {
  // erzeugt komplementären Graph

  private Graph graph;

  public Komplement (Graph g)
    {graph = g;}

  public Graph GraphNeu() {
    int n = graph.AnzahlKnoten();
    Graph gNeu = new Graph(n, false);
    for (int i = 0; i < n - 1; i++)
      for (int j = i + 1; j < n; j++)
        if (! graph.IstAdjazent(i, j))
          gNeu.einfüge(new Kante(i, j));
    String name = graph.Name();
    gNeu.setzName("KOMPLEMENT(" + name + ")");
    return gNeu;
    } // Ende GraphNeu

  } // Ende Komplement
```

Es gilt der

Satz: Genau dann ist U eine Knotenüberdeckung von G = (E, K), wenn C = E\U eine Clique im Komplement G* = (E, K*) ist.

Wie werden Cliquen konstruiert? Wir könnten wieder einen rekursiven Ansatz versuchen, indem wir im

Aufgabe 7: Implementieren Sie die Prozedur *SucheClique()* und testen Sie sie u. a. am Graphen von Bild 8 (Hinweis: 1 Fünfer-, 2 Vierer-, 14 Dreiercliquen).

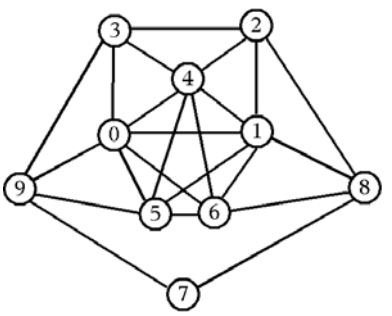

Bild 8: Ramsey-Graph.

```
RAMSEY-GRAPH

Kn: Nachbarn
00: 01 03 04 05 06 09
01: 00 02 04 05 06 08
02: 01 03 04 08
03: 00 02 04 09
04: 00 01 02 03 05 06
05: 00 01 04 06 09
06: 00 01 04 05 08
07: 08 09
08: 01 02 06 07
09: 00 03 05 07

10 Knoten, 23 Kanten

00 01 04 05 06 (5 Aufrufe)
00 03 04 (11 Aufrufe)
00 03 09 (12 Aufrufe)
00 05 09 (19 Aufrufe)
01 02 04 (24 Aufrufe)
01 02 08 (25 Aufrufe)
01 06 08 (33 Aufrufe)
02 03 04 (37 Aufrufe)
07 08 (53 Aufrufe)
07 09 (54 Aufrufe)
```

Bild 9: Die inklusionsmaximalen Cliquen des Ramsey-Graphen (siehe Bild 8, vorige Seite).

Algorithmus SucheUnabh() das Wort *Unabh* durch *Clique* und *nicht-adjazent* durch *adjazent* ersetzen:

SucheClique (Clique, Kand)

```
SOLANGE Kand ≠ leere Menge WIEDERHOLE {
   Entferne p aus Kand und füge p in Clique ein
   KandNeu = Menge der zu p adjazenten Knoten von Kand
   WENN KandNeu = leere Menge DANN Ausgabe: Clique
   SucheClique(Clique, KandNeu) // rekursiver Aufruf
   Entferne p aus Clique
   } // Ende-wiederhole
```

(Genau genommen operierte bereits der Algorithmus sucheUnabh() auf dem Komplement-Graphen.)

Von den holländischen Informatikern Coen Bron und Joep Kerbosch (aus der Schule Edsger Dijkstras) stammt ein Algorithmus, der das naive Verfahren wesentlich verbessert. Sie arbeiten mit drei Mengen, und zwar neben *Clique* (dem schrittweise aufzubauenden vollständigen Teilgraphen) und *Kand* (der Menge der jeweils zur Erweiterung vorgesehenen Knoten) zusätzlich mit einer Menge namens *Erledigt*. Diese enthält die anlässlich der Erweiterung von *Clique* übernommenen Knoten, die nunmehr von der Aufnahme ausgeschlossen sind. Solange die Menge *Erledigt* nicht leer ist, ist die aktuelle – im Aufbau begriffene – Clique noch nicht maximal, vielmehr in einer anderen Clique enthalten (vgl. Bron/Kerbosch, 1973).

SucheClique (Clique, Kand, Erledigt)

```
SOLANGE Kand ≠ leere Menge WIEDERHOLE {
   Entferne p aus Kand und füge p in Clique ein
   KandNeu = Menge der zu p adjazenten Knoten von Kand
   ErledigtNeu = Menge der zu p adjazenten Knoten
         von Erledigt
   WENN KandNeu = leere Menge UND ErledigtNeu =
         leere Menge DANN Ausgabe: Clique
   SucheClique(Clique, KandNeu, ErledigtNeu) //
         rekursiver Aufruf
   Entferne p aus Clique, füge p in Erledigt ein
   } // Ende-wiederhole
```

Aufgabe 8: Aus der Lösung des Cliquenproblems lässt sich eine Lösung des Knotenüberdeckungsproblems gewinnen. Dazu gehen wir wie folgt vor: Wir bilden zunächst das Komplement g* zum gegebenen Graphen g = G(E, K), ermitteln eine maximale Clique C = MaxClique(g*) und gehen schließlich zur Menge U = E\C über (siehe Bild 10 und vergleiche mit Bild 5, S. 85).

```
KOMPLEMENT(PETERSEN-GRAPH)

Kn: Nachbarn
00: 02 03 05 07 08 09
01: 03 04 05 06 08 09
02: 00 04 05 06 07 09
03: 00 01 05 06 07 08
04: 01 02 06 07 08 09
05: 00 01 02 03 06 09
06: 01 02 03 04 05 07
07: 00 02 03 04 06 08
08: 00 01 03 04 07 09
09: 00 01 02 04 05 08

10 Knoten, 30 Kanten

01 03 04 06 07 08
01 02 04 05 06 09
00 02 04 07 08 09
00 02 03 05 06 07
00 01 03 05 08 09
```

Bild 10: Maximale Überdeckungen des Petersen-Graphen via Cliquen.

In JAVA lautet der Bron-Kerbosch-Algorithmus wie folgt:

```
private void sucheClique (ArrayList Clique,
                          ArrayList Kand,
                          ArrayList Erledigt) {
  while (! Kand.isEmpty()) {
    Integer p = (Integer) Kand.remove(0);
    Clique.add(p);
    ArrayList KandNeu =
      Schnitt(Kand, graph.Adjazenz(p.intValue()));
    ArrayList ErledigtNeu =
      Schnitt(Erledigt,
              graph.Adjazenz(p.intValue()));
    if (KandNeu.isEmpty() &&
              ErledigtNeu.isEmpty()) {
      ergebnis += graph.Knotenausgabe(Clique);
      ergebnis += " (" + aufrufe + " Aufrufe)\n";
      } // Ende if
    sucheClique(Clique, KandNeu, ErledigtNeu);
    Clique.remove(p); Erledigt.add(p);
    } // Ende while
  } // Ende sucheClique
```

Das zugehörige Testprogramm liefert die Ausgabe von Bild 9.

Rolf Niedermeier
Jörg Vogel
Michael Fothe
Friedrich-Schiller-Universität Jena
Fakultät für Mathematik und Informatik
07740 Jena

Mirko König
Carl-Zeiss-Gymnasium Jena
Erich-Kuithan-Straße 7
07743 Jena

E-Mail:
{niedermr/vogel/fothe/mkoenig}@minet.uni-jena.de

Das Kapitel „Beispiele für den Unterricht“ (ab Seite 83) wurde freundlicherweise von Rüdeger Baumann zur Verfügung gestellt:

Rüdeger Baumann
Am Fuchsgarten 3
30823 Garbsen

E-Mail: baumann-garbsen@t-online.de

Die JAVA-Programme können über den **LOG-IN-Service** (siehe S. 108) bezogen werden.

Literatur

Arbeitskreis „Bildungsstandards“ der Gesellschaft für Informatik (Hrsg.): Grundsätze und Standards für die Informatik in der Schule – Bildungsstandards Informatik. In: LOG IN, 27. Jg. (2007), Heft 146/147, Beilage.

Breier, N.: Grenzen des Computereinsatzes. In: LOG IN, 10. Jg. (1990), Teil 1: H. 1, S. 43–50; Teil 2: H. 3, S. 37–42.

Bron, C.; Kerbosch, J.: Algorithm 457. Finding all Cliques of an Undirected Graph. In: Communication of the ACM, 16. Jg. (1973), H. 9, S. 575–577.

Cormen, T. H.; Leiserson, C. E.; Rivest, R.; Stein, C.: Algorithmen – Eine Einführung. München: Oldenbourg Wissenschaftsverlag, ^{2}2007.

Diestel, R.: Graphentheorie. Berlin; Heidelberg: Springer, ^{3}2006.

Garey, M. R.; Johnson, D. S.: Computers and Intractability – A Guide to the Theory of NP-Completeness. New York (NY, USA): W. H. Freeman, 1979.

Gasarch, W. I.: The P=?NP Poll. In: ACM SIGACT News, 33. Jg. (2002), H. 2, S. 34–47.

Hopcroft, J. E.; Motwani, R.; Ullman, J. D.: Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie. Reihe „Pearson Studium“. München u. a.: Addison-Wesley, ^{2}2002.

Kerner, I. O.: Harte Nüsse – Die schwierigsten Probleme für den Computer. In: LOG IN, 11. Jg. (1991), H. 3, S. 9–17.

Kleinberg, J.; Tardos, É.: Algorithm Design. Reihe „Pearson Education“. Reading (MA, USA) u. a.: Addison-Wesley, 2005.

Ladner, R. E.: On the structure of polynomial time reducibility. In: Journal of the ACM, 22. Jg. (1975), H. 1, S. 151–171.

Niedermeier, R.; Vogel, J.; Fothe, M.; König, M.: Das Knotenüberdeckungsproblem – Eine Fallstudie zur Didaktik NP-schwerer Probleme (Teil 1). In: LOG IN, 27. Jg. (2007), H. 146/147, S. 53–59.

Papadimitriou, H. C.: Computational Complexity. Reading (MA, USA) u. a.: Addison-Wesley, 1994.

Schwill, A.: Praktisch unlösbare Probleme. In: LOG IN, 14. Jg. (1994), H. 4, S. 16–21.

Wanka, R.: Approximationsalgorithmen – Eine Einführung. Wiesbaden: Teubner, 2006.

Wegener, I.: Komplexitätstheorie – Grenzen der Effizienz von Algorithmen. Berlin; Heidelberg u. a.: Springer, 2003.

Zu beiden Teilen des Beitrags „Das Knotenüberdeckungsproblem“ (siehe auch Teil 1, LOG IN 146/147, S. 53 ff.) wird eine Umfrage veranstaltet, zu der die Autoren alle Leserinnen und Leser herzlich einladen. Die Internetadresse lautet

http://www.minet.uni-jena.de/~mkoenig/npumfrage.html