Wir haben schon verschiedentlich asymptotische Komplexität betrachtet.

Aber bislang immer die Komplexität einzelner Algorithmen.

In diesem Video betrachten wir die Komplexität von algorithmischen Problemstellungen.

Das heißt, was ist die bestmögliche asymptotische Komplexität, die ein Algorithmus für diese Problemstellung überhaupt haben kann.

======

Was das bedeutet, sehen wir uns an einer konkreten algorithmischen Problemstellung an, dem Sortierproblem. Hierfür weiß man sehr gut, was möglich ist und was nicht.

Genauer gesagt die generische Variante, in der nichts über den zu sortierenden Typ bekannt und nur ein Vergleichsoperator gegeben ist. Selection Sort, Bubblesort, Mergesort und Quicksort lösen diese Variante, Bucketsort hingegen löst eine weitaus eingeschränktere Variante, nämlich lexikographische Sortierung von Strings.

Das Beste, was wir bisher gesehen haben, ist n-log-n. Mergesort hat das im Worst Case erreicht.

Ein Algorithmus mit besserer, sogar linearer Komplexität für Sortieren mit paarweisen Vergleichen wäre erst einmal denkbar. Weniger als linear geht natürlich nicht, denn jedes zu sortierende Element muss natürlich mindestens einmal angeschaut werden. Wir werden aber sehen, dass im Worst Case nichts Besseres als n-log-n möglich ist.

Das kann man streng mathematisch beweisen, und das schauen wir uns jetzt an.

======

Bis jetzt haben wir immer *obere* Komplexitätsschranken bewiesen, nämlich für einzelne Algorithmen. Wenn wir nun eine *untere* Schranke beweisen wollen, können wir uns das Leben an einigen Stellen einfacher machen.

Im Gegensatz zum Beweis von oberen Schranken brauchen wir gar nicht alle Anweisungen zu betrachten, nicht einmal alle *Typen* von Anweisungen. Wenn ein Algorithmus schon allein n-log-n Anweisungen von einem Typ benötigt, kann seine Gesamtkomplexität natürlich nicht geringer als n-log-n sein. Wir werden zeigen, dass jeder *auch nur denkbare* Algorithmus für das generische Sortierproblem, also für Sortieren allein mit paarweisen Vergleichen, auf jeden Fall n-log-n Vergleichsoperationen benötigt.

Genauso muss man für untere Worst-Case-Schranken nicht jeden einzelnen möglichen Input berücksichtigen. Wenn ein Algorithmus Komplexität n-log-n schon auf den Permutationen der Zahlen 1 bis n hat, dann kann seine Gesamtkomplexität im Worst Case wieder nicht geringer als n-log-n sein.

Man muss nicht einmal das generische Sortierproblem selbst betrachten. Es reicht, ein algorithmisches Problem zu betrachten, das von jedem Algorithmus, der 1 bis n korrekt sortiert, implizit mit gelöst wird. Konkret könnten wir bei jedem beliebigen Algorithmus die Ergebnisse der einzelnen Vertauschungen mitprotokollieren. Für zwei verschiedene Inputpermutationen kommen dabei zwangsläufig verschiedene Protokolle heraus, denn die Folge der Vertauschungen ergibt genau die Umkehrpermutation. Jeder Algorithmus für das generische Sortierproblem identifiziert nebenher die Inputpermutation.

======

Diesen letzten Punkt kann man sehr gut veranschaulichen durch einen sogenannten *Entscheidungsbaum*. Allerdings sind Entscheidungsbäume für längere Inputsequenzen viel zu groß, wir müssen uns Ausschnitte ansehen. Jeder generische Sortieralgorithmus hat für jede feste Inputgröße klein-n einen Entscheidungsbaum.

Wie gesagt, betrachten wir nur den Fall, dass die Inputsequenz eine Permutation ist, insbesondere schließen wir hier den Fall aus, dass zwei Elemente gleich sein könnten.

Für alle n>4 sind die ersten drei Stufen von Mergesort identisch, denn die Sequenz wird in der Rekursion schrittweise in Teilsequenzen zerlegt, bis jede Teilsequenz nur noch zwei Elemente hat. Erst dann beginnen die Vergleiche. Auf der obersten Stufe werden die beiden Elemente der ersten Teilsequenz miteinander verglichen, auf der zweiten Stufe die der zweiten Teilsequenz, dann die der dritten Teilsequenz und so weiter.

======

Um einen ganzen Durchlauf durch Mergesort inklusive nichttrivialem Durchlauf durch Merge wenigstens für einzelne Permutationen auf einer Folie darstellen zu können, schauen wir uns den Fall n=4 an und hier eben auch nur für zwei Permutationen. Auf den ersten beiden Stufen werden wieder wie eben die beiden Elemente in der ersten

beziehungsweise die beiden Elemente in der zweiten Teilsequenz miteinander verglichen, also zwei Aufrufe von merge.

Alle weiteren Stufen gehören zum dritten und letzten Aufruf von Merge, der die beiden in sich sortierten Teilsequenzen (a[0],a[1]) und (a[3],a[2]) verarbeitet. Jede Inputpermutation induziert genau einen Pfad im Baum von der Wurzel bis zu einem Blatt.

======

Wir haben also jetzt gesehen, dass jeder generische Sortieralgorithmus die Inputpermutation identifiziert. Wir müssen also nur noch einsehen, dass *dieses* algorithmische Problem, nämlich eine gegebene Permutation zu identifizieren, mindestens Größenordnung n-log-n Vergleiche im Worst Case benötigt.

Machen wir uns klar, dass jeder infrage kommende Algorithmus die Inputpermutation schrittweise eingrenzt.

Zu Beginn des Algorithmus noch keine Eingrenzung. Vor dem allerersten Vergleich ist noch jede Inputpermutation möglich. Wir sind an der Wurzel des Entscheidungsbaumes, und mit jedem Vergleich steigen wir im Entscheidungsbaum eine Stufe hinab.

Am Ende des Algorithmus sind wir an einem Blatt des Entscheidungsbaumes angekommen, und die Inputpermutation ist vollständig eingegrenzt.

Wenn zwei Elemente a und b im Algorithmus verglichen werden, dann scheiden entweder alle Permutationen mit a kleiner b oder alle Permutationen mit a größer b aus. Wie gesagt, den Fall a gleich b betrachten wir nicht.

--

Nichtredundant heißt, dass in jedem der beiden Fälle tatsächlich Permutationen ausgeschlossen werden. *Redundante* Vergleiche können wir natürlich aus der Betrachtung herausnehmen.

======

Aus den Mathematikveranstaltungen sollte bekannt sein, dass es genau n-Fakultät verschiedene Permutationen einer n-elementigen Menge gibt.

Und wenn man schrittweise die Menge aller Permutationen jeweils in zwei Teile zerlegt, ist die Tiefe der Zerlegung im Worst Case mindestens logarithmisch in der Anzahl der Permutationen.

======

Die mathematische Auflösung benötigt nur ein paar Schritte.

Einfach die Definition der Fakultät eingesetzt.

Funktionalgleichung des Logarithmus n-minus-1-mal angewandt.

Der Logarithmus ist monoton wachsend, daher sind alle Summanden ab dem mittleren mindestens so groß wie der mittlere, also mindestens Logarithmus von n halbe.

Es gibt natürlich n halbe Summanden ab dem mittleren Summanden.

Noch einmal die Funktionalgleichung des Logarithmus, diesmal auf das Argument n-halbe angewandt.

Ein halb mal Logarithmus zwei ist eine Konstante.

Wenn es wie hier einen Summanden höchster Ordnung gibt, dominiert er bekanntlich die Asymptotik der gesamten mathematischen Funktion.

Und damit ist die untere Schranke n-log-n bewiesen. Leider kennt man nur für sehr wenige algorithmische Problemstellungen eine untere Schranke außer der trivialen linearen.

======

Das generische Sortierproblem war für uns hier nur eine Vorübung. Zentral in diesem Video sind nun algorithmische Problemstellungen, die besonders schwer sind – sehr viel

schwerer als alle bisher in dieser Vorlesung betrachteten. Wir gehen die Beispiele in dieser Liste auf den folgenden Folien einzeln durch.

======

Das Traveling Salesperson Problem, also das Handlungsreisendenproblem, abgekürzt TSP, ist leicht erklärt. In der hier gezeigten sogenannten Euklidischen Variante sind endlich viele Punkte in der Ebene gegeben, und gesucht ist ein geschlossener Streckenzug mit minimaler Euklidischer Gesamtlänge.

======

Um das Rucksackproblem zu illustrieren, nehme ich als Beispiel einen LKW her, auf dem Sie Objekte transportieren wollen, um sie anderswo zu verkaufen.

Der LKW hat ein gewisses maximales Zuladungsgewicht, hier mit groß-G abgekürzt.

Jedes Objekt, das Sie möglicherweise mitnehmen wollen, hat ein gewisses Gewicht.

Jedes Objekt erzielt aber auch einen gewissen Erlös, wenn Sie es mitnehmen.

Wegen der maximalen Zuladung können Sie aber nicht alle Objekte mitnehmen, sondern müssen eine Auswahl treffen.

Aber natürlich wollen Sie die Auswahl nicht irgendwie treffen, sondern so, dass der Gesamterlös aus allen mitgenommenen Objekten möglichst groß ist.

Optimierungsprobleme dieser Art tauchen zuhauf in der Praxis auf. Meist ist die Situation natürlich viel komplexer. Aber da diese Vereinfachung schon schwer ist in dem noch zu bestimmenden Sinne, sind die realen Probleme in der Praxis natürlich erst recht in diesem Sinne schwer.

======

Das nächste Beispiel ist wieder gut visualisierbar. Es taucht regelmäßig bei der Konstruktion von Netzwerken aller Art auf. Gegeben ist ein ungerichteter Graph, jede Kante hat einen Kostenwert, und eine Auswahl von Knoten im Graphen ist ebenfalls gegeben. Die ausgewählten Knoten heißen Terminale.

Zu konstruieren ist ein Baum im Graphen, der alle Terminale verbindet, aber nicht irgendein Baum, sondern ein kostenminimaler. Die Kosten eines Baumes sind gerade die Summe der Kosten aller Kanten im Baum.

Das Steinerbaumproblem ist also eine Verallgemeinerung des MST. Im MST sind eben alle Knoten Terminale. Wie wir wissen, ist das MST sehr effizient lösbar, auch im Worst Case. Diese Verallgemeinerung hingegen ist im Worst Case schwer in dem noch zu bestimmenden Sinne.

======

Noch eine algorithmische Problemstellung, die sehr häufig in komplizierteren Problemstellungen aus der Praxis drinsteckt. Letztere sind natürlich wieder schwer, weil schon dieses Grundproblem schwer ist.

Denken Sie zum Beispiel an eine Menge groß-G von Empfängern.

Jede Servicestation kann eine gewisse Menge an Empfängern bedienen, also eine gewisse Teilmenge von groß-G. Groß-T ist die Menge der Orte, an denen wir tatsächlich Servicestationen einrichten wollen.

Aus Kostengründen wollen wir aber nicht an *allen* möglichen Orten jeweils eine Servicestation platzieren, sondern an *möglichst wenigen*.

So dass aber immer noch jeder einzelne Empfänger von einer der Servicestationen bedient wird.

Diese theoretische Problemstellung gibt es in zwei Varianten: Entweder ist jeder Empfänger bei *genau* einer Servicestation oder bei *wenigstens* einer Servicestation in der Menge der überdeckten Empfänger drin. Letzterer Fall ist natürlich eher der praxisrelevante.

======

Noch ein Beispiel auf Graphen.

Auf *ungerichteten* Graphen, um genau zu sein.

In beiden Varianten ist eine möglichst große Knotenmenge zu bestimmen.

Entweder *alle* Knoten paarweise miteinander verbunden, eine solche Knotenmenge heißt *Clique*.

Oder *alle* Knoten paarweise *nicht* miteinander verbunden, das nennt man dann eine *unabhängige* Knotenmenge.

======

Ein kleines Beispiel zur Illustration. Es gibt genau eine Clique der Größe 4 und keine größeren. Ich zähle sechs Cliquen der Größe 3. Wenn ich mich nicht verguckt habe, gibt es mehrere unabhängige Mengen der Größe 4 und keine größeren. Ich zähle vier unabhängige Mengen der Größe 4, suchen Sie die alle zur Überprüfung Ihres Verständnisses selbst!

======

Jetzt haben wir ein paar Beispiele für schwere algorithmische Problemstellungen gesehen, aber noch überhaupt nicht geklärt, was ein schweres Problem eigentlich ist.

Bevor wir uns eine theoretische Definition antun, erst einmal zur Motivation die praktische Konsequenz. Leider trifft diese Aussage auf fast alle algorithmischen Problemstellungen zu, die man in der Praxis so findet. Beachten Sie die Quantoren: Es gibt nicht *eine* globale Menge von problematischen Beispielen, sondern für jeden *Algorithmus* gibt es problematische Beispiele.

Im Video zu algorithmischen Konzepten sowie in weiterführenden Vorlesungen sehen Sie Methoden, wie man mit diesem Problem mehr oder weniger gut umgehen kann.

======

Nun zur theoretischen Behandlung des Themas.

Mangels Perspektiven sind die theoretischen Informatiker in den sechziger und siebziger Jahren bescheiden geworden. Es wäre schon ein gigantischer Fortschritt, wenn man auch nur für eines dieser schweren Probleme einen Algorithmus finden würde, dessen Worst-Case-Komplexität wenigstens durch ein *Polynom* abgeschätzt werden kann, egal welcher Exponent. Selbst ein Exponent eine Million oder eine Billion würde die ganze theoretische Informatik völlig umkrempeln.

Ein solcher Algorithmus ist seit Jahrzehnten für keines dieser schweren Probleme in Sicht. Die theoretischen Informatiker haben aber ein geniales Konzept zur genaueren Analyse dieser Art von schweren Problemen gefunden. Ausgangspunkt der Überlegung ist: Wenigstens kann man für eine gegebene Lösung in polynomieller Zeit nachprüfen, dass sie tatsächlich eine Lösung ist.

======

Wir sprechen jetzt schon seit einiger Zeit von polynomieller Komplexität. Bevor es weitergeht, müssen wir kurz noch einmal klären, was eigentlich das Argument des Polynoms sein soll.

Diese Frage ist nicht trivial, denn im Video zur asymptotischen Komplexität hatten die abschätzenden Funktionen generell mehrere Parameter. Und bei Algorithmen auf Graphen haben wir auch tatsächlich zwei Parameter einfließen lassen, die Knotenzahl und die Kantenzahl.

Wir brauchen für die folgenden Betrachtungen aber *einen* Parameter, nicht mehrere. Dieser eine Parameter ist nicht schwer zu finden: die Gesamtgröße des Inputs, der sich beispielsweise bei einem Graphenproblem aus Knoten und Kanten nebst ihren Attributen zusammensetzt.

Die Inputgröße hängt davon ab, wie die einzelnen Daten kodiert sind, insbesondere wie Zahlen kodiert sind. Wir gehen davon aus, dass Zahlen ganz normal binär oder meinetwegen zu einer anderen Basis als 2 kodiert sind. Bei ganzen Zahlen ist die Kodierungsgröße einer Zahl dann logarithmisch in ihrer Größe, bei reellen Zahlen logarithmisch in der Genauigkeit und im Exponenten.

Für boolesche Werte, Character und so weiter können wir sicher getrost konstante Kodierungslänge annehmen.

Bei zusammengesetzten Datenstrukturen müssen die Kodierungslängen der einzelnen Elemente natürlich addiert werden.

======

Nach der Klärung der Kodierungslänge benötigen wir noch eine weitere konzeptionelle Vorbereitung, nämlich die Unterscheidung verschiedener Klassen von algorithmischen Problemstellungen je nach der Art ihres Outputs.

Das ist der einfachste Fall: Entscheidungsproblem ist alles, wo am Ende ein Ja oder ein Nein herauskommen soll – eine Entscheidung eben, genauer: eine binäre Entscheidung.

Oft gibt es mehrere Lösungen, und es reicht es aus, *irgendeine* Lösung zu finden, egal welche, oder die Antwort zu bekommen, dass es keine Lösung gibt. Letzteres natürlich nur, wenn es wirklich keine Lösung gibt.

Denken Sie etwa an die Erstellung von Stundenplänen für Schulen und Universitäten. Die Nebenbedingungen sind so eng und unübersichtlich, dass man schon froh sein kann, wenigstens *irgendeine* Lösung zu finden – oder eben herauszufinden, dass die Nebenbedingungen keine Lösung zulassen.

Manchmal will man statt *einer* Lösung *alle* oder zumindest *viele* Lösungen finden, wenn es denn überhaupt mehrere Lösungen gibt. Das ist zum Beispiel der Fall, wenn man Kriterien zur Auswahl der besten Lösung nicht so ohne weiteres in einen Algorithmus packen kann, etwa weiche, subjektive Kriterien oder persönliche Kriterien. Dann möchte man hinterher anhand dieser Kriterien die beste Lösung aus einer Liste von Optionen auswählen. Denken Sie etwa an Fahrplanauskunft für Bahnverkehr.

In einem anderen Fall möchte man unbedingt *alle* Lösungen oder eine *repräsentative* Auswahl von Lösungen haben, nämlich immer dann, wenn man statistische Betrachtungen über die Menge aller Lösungen anstellen möchte.

Aber sehr oft möchte man nach einem numerischen Kriterium tatsächlich eine optimale oder zumindest eine möglichst gute Lösung vom Algorithmus berechnen lassen. Kürzeste Pfade und minimal spannende Bäume waren Beispiele aus anderen Videos, das TSP und das Steinerbaumproblem sind Beispiele auf früheren Folien dieses Videos.

======

In einem Konstruktions- oder Optimierungsproblem steckt immer auch ein Entscheidungsproblem, das wir das *kanonische* Entscheidungsproblem zu diesem Konstruktions- beziehungsweise Optimierungsproblem nennen.

Im kanonischen Entscheidungsproblem für ein *Konstruktions*problem geht es einfach allm die Frage, ob eine Lösung existiert oder nicht. Die eigentlichen Lösungen sind beim kanonischen Entscheidungsproblem uninteressant. Beachten Sie für das Folgende: Wenn ein Algorithmus das Konstruktionsproblem löst, hat er trivialerweise nebenher auch das kanonische Entscheidungsproblem gelöst.

Im kanonischen Entscheidungsproblem für ein *Optimierungs*problem ist stattdessen zu entscheiden, ob es eine Lösung gibt, die eine gewisse Mindestgüte hat. Diese Mindestgüte ist zusätzlich vorgegeben, auf der Folie ist sie mit k bezeichnet. Wenn Kosten zu minimieren sind, dann heißt das, dass die Güte der Lösung *höchstens* gleich k ist. Sind hingegen Profite zu maximieren, soll die Güte der Lösung *mindestens* gleich k sein.

Analog zu Konstruktionsproblemen gilt auch hier (und ist ebenfalls wichtig für alles Folgende): Wenn ein Algorithmus das Optimierungsproblem löst, hat er trivialerweise nebenher auch das kanonische Entscheidungsproblem gelöst.

Warum beschäftigen wir uns überhaupt mit kanonischen Entscheidungsproblemen? Nun, wir sind ja gerade auf dem Weg, schwere Probleme zu untersuchen. Es ist einsichtig, dass Entscheidungsprobleme sich leichter analysieren lassen als andere Probleme, weil wenigstens der Output einfach gestrickt ist.

Und die Idee ist nun: Wenn schon das kanonische Entscheidungsproblem schwer ist, dann ist das zugehörige Konstruktions- beziehungsweise Optimierungsproblem ebenfalls schwer, denn, wie wir gesagt haben, wird das kanonische Entscheidungsproblem ja nebenher mitgelöst, kann also höchstens genauso schwer sein.

======

Bevor wir unser eigentliches Ziel ins Auge fassen, brauchen wir *noch* eine konzeptionelle Voraussetzung. Sie kennen Zertifikate aus anderen Kontexten.

Zertifikate stehen tatsächlich in engem Zusammenhang mit Entscheidungsproblemen. Ganz allgemein und abstrakt ist ein Zertifikat bezogen auf einen Input, bei dem die Antwort Ja lautet.

Zum Konzept eines Zertifikats gehört ein Prüfalgorithmus oder Zertifizierungsalgorithmus, der einen Ja-Input und das zugehörige Zertifikat hernimmt und eben mit Hilfe dieses Zertifikats bestätigt, dass der Ja-Input tatsächlich ein Ja-Input ist.

Wird der Prüfalgorithmus mit einem Nein-Input und einem angeblichen Zertifikat aufgerufen, dann gibt es keine Anforderung, was dieser Algorithmus macht: Er darf Ja oder Nein zurückliefern, er darf in eine Endlosschleife geraten oder auch abstürzen. Endlosschleife oder Absturz ist hier für unsere theoretischen Betrachtungen ok, in der Praxis darf das natürlich aus nachvollziehbaren Gründen nicht passieren.

======

Um das Konzept namens Zertifikat besser zu verstehen, betrachten wir zwei Beispiele aus der Praxis.

An viele Arten von Information ist eine Checksumme angehängt. Ein konkretes Beispiel ist die IBAN, die sich in Deutschland aus der Länderkennung DE, der Bankleitzahl und der Kontonummer nebst Auffüllung mit Nullen zusammensetzt, und auf Position 3 und 4, direkt hinter der Länderkennung, steht eine Checksumme aus zwei Ziffern.

Wenn die IBAN inklusive Checksumme korrekt ist, dann liefert der Prüfalgorithmus Ja. Ist die IBAN inklusive Checksumme *nicht* korrekt, dann liefert er Ja oder Nein. Die Checksumme hat einen Wert zwischen 02 und 98, das heißt, statistisch in einem von 97 Fällen lautet die Antwort Ja, wenn Nein die richtige Antwort wäre.

Ein anderes Beispiel aus der IT-Sicherheit ist der öffentliche Schlüssel für elektronisch signierte Botschaften. Wenn die Botschaft tatsächlich vom richtigen Absender stammt und auch korrekt verschlüsselt ist, dann liefert der Prüfalgorithmus ein Ja auf Basis des öffentlichen Schlüssels. Falls nicht, dann wird praktisch immer ein Nein geliefert, aber in einer astronomisch kleinen Zahl von Fällen eben fälschlich ein Ja.

Diese beiden Beispiele zeigen, dass ein Zertifikat nach einer beliebigen Logik konzipiert sein kann. Bei kanonischen Entscheidungsproblemen reicht praktisch immer eine ganz einfache Logik schon aus. Die Logik ist: Jede Lösung eines Konstruktionsproblems ist ein geeignetes Zertifikat für das zugehörige kanonische Entscheidungsproblem; bei einem Optimierungsproblem ist jede Lösung, die die angegebene Mindestgüte hat, ein geeignetes Zertifikat. Was heißt das in beiden Fällen konkret?

======

Sehen wir uns das beispielhaft am TSP an.

Rundreisen können in verschiedener Form kodiert von einem Algorithmus ausgegeben werden.

Zum Beispiel als Permutation der *n* Punkte. Wenn eine Rundreise also als geordnete Sequenz von Punkten gegeben ist, wird einfach abgeprüft, dass diese Sequenz tatsächlich genau Länge n hat, dass jeder einzelne Punkt in der Sequenz vorkommt und dass die Sequenz keine Duplikate enthält. Natürlich muss auch die Mindestgüte noch überprüft werden.

Eine Rundreise kann aber beispielsweise auch als Auswahl der Kanten kodiert sein, aus denen die Rundreise sich zusammensetzt. Diverse Algorithmen für das TSP setzen die Rundreise tatsächlich aus der Menge aller möglichen Kanten zusammen.

In diesem Fall prüfen wir, dass genau eine Kante in jeden Punkt hineinzeigt und genau eine Kante aus jedem Punkt herauszeigt.

Und zusätzlich, ganz wichtig: dass die Auswahl eine einzige Rundtour auf allen Punkten ist und nicht in mehrere diskjunkte Rundtouren zerfällt.

======

In diesem Beispiel zeigt tatsächlich genau eine Kante in jeden Punkt hinein und genau eine Kante aus jedem Punkt heraus. Wir müssen also unbedingt zusätzlich prüfen, wenn wir an irgendeinem Punkt starten und entlang der ausgewählten Kanten schrittweise von Punkt zu Punkt gehen, dann kommen wir nicht früher als nach *n* Schritten zurück zum Startpunkt, wenn *n* wieder die Anzahl der Punkte ist.

======

Jetzt sind wir endlich soweit, dass wir zum Kern unseres Themas vorstoßen können.

Zunächst eine – hoffentlich – einfach zu verstehende Klasse von algorithmischen Problemstellungen, nämlich alle, für die es einen Algorithmus gibt, dessen Komplexität im Worst Case durch ein Polynom beliebigen Grades nach oben abschätzbar ist.

Jetzt eine auf den ersten Blick ähnliche, aber konzeptionell doch sehr unterschiedliche Klasse: Es muss keinen polynomiellen Algorithmus geben, der das Problem löst, sondern es reicht, wenn es ein Zertifikat mit einem Prüfalgorithmus gibt, dessen Komplexität im Worst Case durch ein Polynom nach oben abschätzbar ist.

NP steht für nichtdeterministisch polynomiell. Diese Begriffsbildung erwähne ich nur am Rande, wir werden sie hier nicht erklären und motivieren, das würde viel zu viel Aufwand erfordern und viel zu weit führen. Hier soll reichen, dass es grundlegend verschiedene Zugänge zu dem Thema gibt, und die Begriffsbildung stammt eben aus einem anderen Zugang.

Aus den Definitionen von $\mathcal{P}$ und $\mathcal{NP}$ ergibt sich sofort, dass $\mathcal{P}$ eine Teilmenge von $\mathcal{NP}$ ist, denn ein Algorithmus, der für jeden Ja-Input tatsächlich ein Ja liefert, überprüft damit auch, dass ein Ja-Input wirklich ein Ja-Input ist, und zwar mit leerem Zertifinur kat.

An dieser Frage beißen die theoretischen Informatiker sich schon seit mehr als vierzig Jahren die Zähne aus. Manche argwöhnen sogar, dass diese Frage unentscheidbar im Sinne von Gödels Unvollständigkeitssatz ist.

======

Jetzt brauchen wir nur noch ein einziges weiteres Grundkonzept, nämlich was es heißt, dass ein algorithmisches Problem X auf ein anderes algorithmisches Problem Y *reduzierbar* ist.

Grob gesprochen heißt das, dass man *X* lösen kann, indem man einen Input von *X* mit vernachlässigbarer Komplexität in einen Input von *Y* transformiert, dann *Y* löst und das Ergebnis mit vernachlässigbarer Komplexität wieder auf *X* zurücktransformiert.

Wenn der Aufwand für die beiden Transformationen wirklich wie gesagt vernachlässigbar ist, dann kann *Y* zwangsläufig keine geringere Komplexität haben als *X*. Denn jede Komplexität, die ein Algorithmus für *Y* haben kann, hat dann auch der daraus resultierende Algorithmus für *X*.

Um dieses abstrakte Konzept zu verstehen, werden wir auf den folgenden Folien zwei illustrative Beispiele betrachten. In beiden Beispielen ist *Y* das Kürzeste-Pfade-Problem in der Variante, dass die Kanten nichtnegativ sind und ein einzelner kürzester Pfad von einem Startknoten zu einem Zielknoten zu finden ist. Diese Variante ist besonders schön, da sie mit dem Algorithmus von Dijkstra extrem effizient gelöst werden kann.

======

Das erste Beispiel stammt aus der Biologie und ist links gezeigt. Die genetische Information in einer DNA-Sequenz lässt sich bekanntlich abstrahieren zu einem String über einem Alphabet mit vier Buchstaben. Die Ähnlichkeit zweier DNA-Sequenzen ist in der einfachsten Variante definiert als die maximale Länge eines gemeinsamen Teilstrings, der durch Streichung einzelner Zeichen aus jedem der beiden Strings entsteht.

Ein Input besteht eben aus den beiden Strings und wird wie rechts gezeigt in einen gerichteten Graphen mit strikter Gitterstruktur transformiert. Eine Diagonale wird gezogen, wenn das Zeichen links und das Zeichen oben identisch sind. Alle Kanten sollen Länge 1 haben. Der kürzeste Pfad von links oben nach rechts unten hat die maximale Anzahl Diagonalen, also Übereinstimmungen. Der gemeinsame String links ist die Folge der verwendeten Diagonalen rechts, das ist die Rücktransformation.

======

Ein weiteres Beispiel stammt aus dem automatischen Textsatz. Gegeben ist der Text eines Absatzes, zu bestimmen sind Zeilenumbrüche, so dass die einzelnen Zeilen in Summe möglichst wenig hässlich sind. Der Einfachheit halber berücksichtigen wir nicht Silbentrennung mitten im Wort.

Der Graph hat einen Knoten für jedes Wort im Text. Eine Kante zeigt von einem Wort *A* auf ein später im Text auftretendes Wort *B*, wenn *A* und *B* am Anfang zweier direkt untereinander stehender Zeilen stehen könnten. Mit anderen Worten: Der Teiltext von A *in*klusive *A* bis *B ex*klusive *B* passt in eine Zeile. Die Länge dieser Kante gibt an, wie hässlich diese Zeile wäre. Ein kürzester Pfad minimiert also die Summe der Hässlichkeitswerte. Da nicht von vornherein klar ist, mit welchem Wort die letzte Zeile beginnt, wird noch ein zusätzliches künstliches Endwort benötigt, das den Anfang der nächsten Zeile nach dem Absatz simuliert.

======

Nach diesen beiden hoffentlich illustrativen Beispielen nun noch einmal die Zusammenfassung.

Wir haben zwei Probleme auf kürzeste Pfade reduziert.

Dazu haben wir jeweils aus dem Input einen gerichteten Graphen mit nichtnegativen Kantenlängen, einem Start- und einem Zielknoten gebastelt.

Und wir haben jeweils gesehen, wie aus einem kürzesten Pfad dann eine optimale Lösung für das Ausgangsproblem konstruiert werden kann. Diese Rücktransformation war in beiden Beispielen offensichtlich und sehr leicht, das ist natürlich nicht immer so.

Wir haben ja gesagt, dass wir uns der Einfachheit halber auf Entscheidungsprobleme konzentrieren. Hier ist der Grund, warum Entscheidungsprobleme einfacher sind: Es gibt nur zwei mögliche Rücktransformationen von Boolesch auf Boolesch: entweder wird Ja auf Ja und Nein auf Nein transformiert oder eben Ja auf Nein und Nein auf Ja. In beiden Fällen gibt es nichts zu überlegen.

Jetzt ein Schritt weiter:

Eine Reduktion von einem Problem *X* auf ein anderes Problem *Y* nennen wir *polynomiell*, wenn beide Transformationen in polynomieller Komplexität in der Größe des ursprünglichen Inputs für Problem *X* sind. Für unsere grundsätzliche theoretische Betrachtung hier, wo es nur um polynomiell oder nichtpolynomiell geht, ist polynomielle Komplexität der beiden Transformationen tatsächlich vernachlässigbar.

Auch hier brauchen wir uns natürlich um die Rücktransformation keine Gedanken machen, wenn wir uns auf Entscheidungsprobleme konzentrieren.

======

So, jetzt haben wir endgültig alle Bestandteile zusammen und können die schweren Probleme, von denen immer die Rede ist, klassifizieren.

Wir sagen, Problem *X* in $\mathcal{NP}$ ist $\mathcal{NP}$*-vollständig*, englisch $\mathcal{NP}$*-complete*, wenn sich jedes einzelne, *jedes einzelne* Problem in $\mathcal{NP}$ polynomiell auf *X* reduzieren lässt.

Jedes einzelne Problem in $\mathcal{NP}$, also zunächst einmal eine atemberaubende Anforderung.

Bis jetzt ist überhaupt nicht klar, ob es auch nur ein einziges $\mathcal{NP}$-vollständiges Problem gibt. Dass es sogar sehr viele gibt, davon werden wir im Rest des Videos einen Eindruck erhalten.

Die theoretisch wie praktisch spannende Frage, ob $\mathcal{P}=\mathcal{NP}$ ist, können wir mit Hilfe der Klasse $\mathcal{NPC}$ übrigens umformulieren.

Wenn $\mathcal{P}=\mathcal{NP}$ ist, dann sind natürlich auch alle $\mathcal{NP}$-vollständigen Probleme in $\mathcal{P}$. Umgekehrt: Gibt es einen polynomiellen Algorithmus auch nur für ein einziges $\mathcal{NP}$-vollständiges Problem, ein einziges, dann sind *alle* Probleme in $\mathcal{NP}$ polynomiell lösbar, denn *alle* Probleme in $\mathcal{NP}$ sind auf dieses $\mathcal{NP}$-vollständige Problem polynomiell reduzierbar.

======

Jetzt lernen wir unser erstes $\mathcal{NP}$-vollständiges Problem kennen.

Es geht um die logische Erfüllbarkeit von Schaltkreisen mit beliebigen logischen Gattern, also zum Beispiel AND und OR oder NAND und NOR, was auch immer. Und natürlich auch die Negation.

Genauer gesagt geht es um Schaltkreise mit genau einem Ausgangspin, denn wir fokussieren ja auf Entscheidungsprobleme.

Das Entscheidungsproblem ist einfach zu formulieren: Können wir für jeden Eingangspin ein Signal 0 oder 1 so auswählen, dass am Ausgangspin das Signal 1 ankommt. Wir sagen, eine solche Auswahl ist eine *erfüllende Belegung* der Eingangspins.

Für einen gegebenen Schaltkreis und eine erfüllende Belegung können wir ein polynomiell nachprüfbares Zertifikat ganz einfach definieren, nämlich für jedes Gatter das Signal am Ausgangspin dieses Gatters. Diese Zuordnung von Bits zu Gattern, zum Beispiel als Boolescher Array, wenn die Gatter durchnummeriert sind, ist dann das Zertifikat.

Schrittweise, beginnend mit den Eingangspins, kann dann Gatter für Gatter geprüft werden, ob das tatsächliche Signal am Ausgangspin des Gatters tatsächlich das ist, das im Zertifikat für dieses Gatter steht. Der Ausgangspin des letzten Gatters ist der Ausgangspin des ganzen Schaltkreises.

Auf den nächsten Folien sehen wir, dass das Problem, ob es für einen Schaltkreis eine erfüllende Belegung gibt, tatsächlich $\mathcal{NP}$-vollständig ist.

======

Um das zu beweisen, nehmen wir uns irgendein Problem in $\mathcal{NP}$ her und zeigen, wie es polynomiell auf Circuit-Satisfiability reduziert werden kann.

Wenn also Problem *X* in $\mathcal{NP}$ ist, dann gibt es nach Definition der Klasse $\mathcal{NP}$ für jeden Ja-Input ein Zertifikat.

Und es gibt einen Algorithmus, der für jeden Ja-Input nebst Zertifikat in polynomieller Zeit bestätigt, dass er tatsächlich ein Ja-Input ist.

Wir müssen noch einmal genauer als bisher fassen, was es heißt, dass ein Algorithmus polynomiell ist. Nun, die Zeit, die ein Algorithmus auf einem Computer benötigt, wird in Taktzyklen gemessen. Polynomielle Laufzeit heißt also: polynomiell viele Taktzyklen.

======

Ein Computer ist ja im Grunde einfach ein digitales, synchrones Schaltwerk. Die Veränderung der Ladungen in einem Schaltwerk innerhalb eines Taktzyklus ist bekanntlich äquivalent zum Durchlauf durch ein Schaltnetz, in dem jedes Speicherbit des Schaltwerks sowohl ein Eingangsbit als auch ein Ausgangsbit ist. Das Ausgangssignal eines Speicherbits in *einem* Taktzyklus ist das Eingangssignal am selben Speicherbit im *nächsten* Taktzyklus.

Wir gehen nicht von einem bestimmten Computer aus und wissen daher nicht, wie groß er als Schaltnetz ist und wie groß das Teilnetz ist, das tatsächlich für den Durchlauf des Algorithmus verwendet wird. Dieses Teilnetz muss nicht polynomiell groß sein, es kann größer sein.

Aber: Für dieses Teilnetz gibt es ein äquivalentes Schaltnetz, das nur polynomiell groß ist, denn jede logische Formel lässt sich durch eine polynomiell große Anzahl von logischen Gattern darstellen. Wir lösen uns also jetzt vom Computer und betrachten für jeden Taktzyklus des Algorithmus ein für diesen Taktzyklus spezifisches, polynomiell großes Schaltnetz.

Wir können uns jetzt im Geiste vorstellen, dass alle diese Schaltnetze aneinander gehängt werden. Die Verbindung zwischen zwei Schaltnetzen ist eine Kopie aller Speicherbits. Die Speicherbits sind ja sowohl die Ausgangsbits *eines* Schaltnetzes und zugleich auch die Eingangsbits des *nächsten* Schaltnetzes, das passt also. Polynomiell viele Schaltnetze, jedes von Ihnen polynomiell groß, ergeben ein polynomiell großes Schaltnetz.

Eine wichtige Modifikation im Design des Schaltnetzes: Wir legen noch zusätzlich fest, dass diejenigen Eingabepins, die den Input selbst kodieren, fest verdrahtet sind. Das heißt, es liegt bei jeder Belegung immer dasselbe Signal an, nämlich das Signal gemäß Kodierung des Inputs. Eine Belegung der Eingangspins setzt also nur die Werte, die das Zertifikat kodieren.

======

Hier sehen wir noch einmal bildlich die Konkatenation zweier Schaltkreise, die durch zwei aufeinanderfolgende Taktzyklen induziert sind. Die Ausgangspins des ersteren Schaltkreises sind die Eingangspins des nächsten. Nur diejenigen Speicherbits, die vom Algorithmus zum Zwischenspeichern von Information zwischen dem i-ten und dem (i+1)-ten Taktzyklus verwendet werden, werden wie hier gezeigt in die Konkatenation integriert.

======

Und so sieht das Gesamtergebnis aus. Der Schaltkreis ist jetzt nur noch schematisch dargestellt. Alle einzelnen Schaltkreise mit den Speicherbits dazwischen bilden zusammen den dreieckig dargestellten Gesamtschaltkreis.

======

Auf dieser Folie werden wir jetzt fertig mit dem Beweis, dass Circuit-SAT $\mathcal{NP}$-vollständig ist.

Zur Erinnerung: Wir haben uns ein Problem *X* aus $\mathcal{NP}$ und einen Ja-Input *I* von *X* hergenommen und daraus einen polynomiell großen Schaltkreis konstruiert, der *I* mit Hilfe des Zertifikats für *I* überprüft.

Da *I* ein Ja-Input ist, kommt bei der Überprüfung natürlich Ja heraus. Das ist einfach die Definition der Klasse $\mathcal{NP}$ und des Zertifikats.

Umgekehrt können wir Folgendes feststellen: Wenn dieser Schaltkreis erfüllbar ist, wenn es also eine Belegung der Eingangspins mit Werten gibt, so dass am Ausgangspin 1 herauskommt ...

... dann muss der Input ein zertifizierter Ja-Input sein, denn: Der Input ist in denjenigen Eingangspins kodiert, die nun einmal den Input kodieren sollen. Und irgendwelche Signale liegen an den Eingangspins, die das Zertifikat kodieren sollen. Und letztere bilden tatsächlich ein Zertifikat, ob es so gedacht war oder nicht.

Durch diese Äquivalenz zwischen Ja-Input und Erfüllbarkeit des so konstruierten Schaltkreises ist nun endlich bewiesen, dass jedes Problem X in $\mathcal{NP}$ sich auf Circuit-SAT polynomiell reduzieren lässt. Circuit-SAT ist also $\mathcal{NP}$-vollständig.

======

Jetzt kommen wir zum zweiten $\mathcal{NP}$-vollständigen Entscheidungsproblem. Wir werden dafür ausnutzen, dass wir mit Circuit-SAT schon ein erstes $\mathcal{NP}$-vollständiges Entscheidungsproblem haben.

Das zweite Problem ist das sogenannte Satisfizierbarkeitsproblem oder kurz SAT.

Gegeben ist eine beliebige Boolesche Formel, wieder mit beliebigen binären Booleschen Operatoren, also AND, OR, NAND, NOR, und so weiter, sowie die Negation.

Analog zu Circuit-SAT ist das Ergebnis Ja genau dann, wenn die Werte der Variablen so gewählt werden können, dass die gesamte Formel wahr wird.

Ebenfalls analog zu Circuit-SAT ist auch das Zertifikat: Es enthält für jedes Vorkommen eines binären Operators in der Formel den Wahrheitswert, der bei diesem Operator herauskommt.

Der Beweis, dass SAT $\mathcal{NP}$-vollständig ist, wird jetzt sehr kurz und einfach sein.

======

Wir nehmen uns einen beliebigen Input für das $\mathcal{NP}$-vollständige Problem Circuit-SAT her.

Ein Schaltkreis ist ja praktisch identisch mit der ihn beschreibenden, logisch äquivalenten Booleschen Formel. Die Generierung dieser Booleschen Formel aus dem Schaltkreis heraus ist linear, also insbesondere polynomiell.

Und jetzt kommt's: Wir haben schon gesehen, dass jedes Problem in $\mathcal{NP}$ sich polynomiell auf Circuit-SAT reduzieren lässt. Jetzt haben wir außerdem gesehen, dass Circuit-SAT sich polynomiell auf SAT reduzieren lässt. Also lässt sich jedes Problem in $\mathcal{NP}$ polynomiell auf SAT reduzieren, nämlich transitiv über Circuit-SAT.

======

Das nächste Entscheidungsproblem ist eine eingeschränkte Variante von SAT, in der nur eine bestimmte Art von Formeln Input sein kann. Dass SAT $\mathcal{NP}$-schwer ist, heißt aber noch lange nicht, dass auch eine solche Einschränkung $\mathcal{NP}$-schwer ist.

Die Einschränkung besteht darin, dass die Inputformel in konjunktiver Normalform ist und dass keine Klausel mehr als drei Literale enthält.

Wie beim allgemeinen SAT ist auch bei 3-CNF die Ausgabe Ja genau dann, wenn es mindestens eine erfüllende Variablenbelegung gibt.

Wenn ein Problem wie SAT in $\mathcal{NP}$ ist, muss natürlich auch jede Einschränkung wie 3-CNF in $\mathcal{NP}$ sein.

Auf der nächsten Folie nun der Beweis, dass 3-CNF $\mathcal{NP}$-vollständig ist.

======

Auch dieser Beweis ist wieder recht kurz.

Wir nehmen uns irgendeinen beliebigen Input für SAT her.

Aus der Technischen Informatik wissen Sie, dass man jede Boolesche Formel in konjunktive Normalform überführen kann. Die Klauseln können aber natürlich mehr als drei, sogar beliebig viele Literale enthalten.

Schritt für Schritt zerlegen wir nun jede Klausel, die mehr als drei Literale enthält, in zwei kleinere, bis alle Klauseln maximal drei Literale haben. Selbstverständlich machen wir nur Äquivalenztransformationen, was wir dann aber erst noch einsehen müssen.

Der Transformationsschritt ist eigentlich recht einfach, aber wir benötigen eine zusätzliche Boolesche Variable.

Jetzt können wir die Klausel durch zwei ersetzen, indem wir alle bisherigen Literale auf die beiden Klauseln verteilen. Und zwar so, dass eine Klausel *K''* nur noch drei Literale hat und die andere Klausel *K'* kürzer als die ursprüngliche Klausel *K* ist.

Betrachten wir zuerst eine Belegung, die die ursprüngliche Klausel *K* erfüllt. Wenn nun eines der ersten *n*-minus-2 Literale wahr ist, dann setzen wir *x* auf nein, und beide Klauseln *K'* und *K''* sind erfüllt. Ansonsten ist eines der Literale Nummer *n*-minus-1 oder *n* wahr, also setzen wir *x* auf wahr, und wieder sind beide Literale *K'* und *K''* erfüllt.

Umgekehrt betrachten wir jetzt eine Belegung, die die ursprüngliche Klausel *K nicht* erfüllt. Dann können wir *x* auf wahr oder falsch setzen, egal, eine der beiden Klauseln *K'* oder *K''* ist nicht erfüllt. Die Zerlegung von *K* in *K'* und *K''* ist also tatsächlich eine Äquivalenzumformung.

======

Zum Abschluss betrachten wir beispielhaft eine algorithmische Problemstellung, die nichts mit Boolescher Logik zu tun hat, sondern eines der eingangs beispielhaft aufgelisteten „schweren“ Probleme ist. Gegeben ist ein ungerichteter Graph, gesucht ist eine größte Clique darin.

======

Zuerst schreiben wir die formale Definition der Problemstellung auf.

Das Optimierungsproblem, also eine Clique maximaler Größe zu finden, heißt MAX-CLIQUE, das zugehörige kanonische Entscheidungsproblem heißt einfach nur CLIQUE.

Wie immer, wenn es eigentlich um ein Optimierungsproblem geht, wird der Input des zugehörigen kanonischen Entscheidungsproblems ergänzt um eine Zahl k.

Und gesucht ist nicht die größte Clique, sondern eben die Antwort auf die Entscheidungsfrage, ob es eine Clique mindestens der Größe *k* gibt.

Jede Clique der Größe *k* oder größer ist ein Zertifikat, das sich natürlich in polynomieller Zeit überprüfen lässt.

Jetzt also der Beweis, dass dieses Problem auch $\mathcal{NP}$-vollständig ist.

======

Wir haben eben gesehen, dass 3-CNF $\mathcal{NP}$-vollständig ist. Dann ist natürlich erst recht die Verallgemeinerung von 3-CNF $\mathcal{NP}$-vollständig, in der die Klauseln beliebig viele Literale enthalten dürfen. Mit anderen Worten, SAT eingeschränkt auf Formeln in allgemeiner konjunktiver Normalform ist $\mathcal{NP}$-vollständig.

Zur Reduktion dieses $\mathcal{NP}$-vollständigen Problems auf CLIQUE müssen wir aus jedem solchen Input für SAT, also für jede beliebige Formel in konjunktiver Normalform, einen Input für das CLIQUE-Problem basteln, also einen ungerichteten Graphen und eine Zahl *k*. Zuerst der Graph.

Jedem Vorkommen eines Literals in einer Klausel entspricht ein Knoten im zu konstruierenden Graphen. Wir haben also so viele Knoten, wie es insgesamt Literale in der Formel gibt. Wenn dasselbe Literal zweimal vorkommt, also dieselbe Boolesche Variable zweimal negiert oder zweimal unnegiert, dann sind das auch zwei Knoten.

Fast alle Paare dieser Knoten sind jeweils durch eine Kante miteinander verbunden. Es gibt nur eine Ausnahme: Wenn zwei Literale aus verschiedenen Klauseln dieselbe Boolesche Variable sind, aber einmal negiert und einmal unnegiert, dann sind die beiden zugehörigen Knoten *nicht* durch eine Kante verbunden.

Hier ist ein kleines Beispiel, natürlich nur ausschnittweise. Eine durchgezogene Strecke zeigt an, dass die beiden Endknoten durch eine Kante verbunden sind, bei einer gestrichelten Linie wird keine Kante eingefügt. Der Knoten links enthält dieselbe Variable wie die Knoten oben und rechts, aber negiert. Daher gibt es keine Kante zwischen dem linken und dem oberen beziehungsweise rechten.

======

Wir müssen noch die Zahl *k* festlegen.

Die Zahl *k* ist einfach die Zahl der Klauseln in der hergenommenen Formel.

In dieser Beobachtung manifestiert sich nun, dass der konstruierte Input zum Cliquenproblem äquivalent zur ursprünglichen Inputformel ist und somit die Reduktion des Problems, ob eine Formel in konjunktiver Normalform erfüllbar ist, auf das Cliquenproblem korrekt ist.

Zuerst nehmen wir an, dass der konstruierte Graph tatsächlich eine Clique der Größe *k* aufweist. Dann setzen wir alle Literale, die zu den Knoten der Clique gehören, auf wahr. Da keine Kante zwischen einer negierten und einer unnegierten Variable existiert, kann damit keine Variable auf wahr und falsch zugleich gesetzt worden sein, die Belegung ist also konsistent und lässt sich natürlich auch konsistent auf alle anderen Literale erweitern. Da *k* die Anzahl der Klauseln ist, sind alle Klauseln erfüllt.

Umgekehrt nehmen wir jetzt an, dass eine erfüllende Belegung für die Formel vorliegt. Dann wählen wir aus jeder Klausel ein Literal aus, das wahr ist. Darunter kann natürlich dieselbe Variable nicht einmal negiert und einmal unnegiert vorkommen, das heißt, die zugehörigen Knoten sind paarweise durch Kanten verbunden, also eine Clique mit so vielen Knoten, wie es Klauseln gibt. Und das war gerade *k*.

======

Damit beenden wir unseren kurzen Blick auf die Anfangsgründe der Komplexitätstheorie. Wir fassen unser grundsätzliches Vorgehen beim Beweis, dass algorithmische Probleme $\mathcal{NP}$-vollständig sind, noch einmal zusammen.

Wir haben eine allererste algorithmische Problemstellung als $\mathcal{NP}$-vollständig identifiziert. Das war Circuit-SAT.

Diese Problemstellung kann man nutzen, um andere Problemstellungen als $\mathcal{NP}$-vollständig zu klassifizieren. Wir haben das mit SAT gemacht, es gibt natürlich noch viele andere.

Jede algorithmische Problemstellung, die so als $\mathcal{NP}$-vollständig bewiesen werden konnte, kann ihrerseits verwendet werden, um weitere Probleme als $\mathcal{NP}$-vollständig zu beweisen. Zum Beispiel haben wir SAT verwendet, um zu beweisen, dass 3-CNF und CLIQUE $\mathcal{NP}$-vollständig sind.

Und diese kann man dann auch wieder verwenden, um für noch weitere algorithmische Problemstellungen zu beweisen, dass sie $\mathcal{NP}$-vollständig sind. Insgesamt ergibt sich so etwas wie ein Wurzelbaum, dessen Knoten algorithmische Probleme sind, die Wurzel ist das erste $\mathcal{NP}$-vollständige Problem, und eine Kante besagt, dass ein Problem auf das andere reduziert werden konnte.

Wir hatten zu Beginn ein paar algorithmische Problemstellungen kurz angesprochen und behauptet, dass sie alle $\mathcal{NP}$-vollständig sind. Wir können das hier nicht beweisen, diese Probleme sind weiter unten im Wurzelbaum, und die Beweise sind auch wesentlich länger und komplexer als die hier präsentierten. Bei Interesse ist es aber kein Problem, diese Beweise im Internet zu finden.

======