Komplexitätstheorie

Thorsten Alten

Institut für Informatik, Universität Potsdam, 14482 Potsdam

Zusammenfassung *Dieser Artikel gibt eine kurze Einführung in die Grundlagen der Komplexitätstheorie. Er dient als Ergänzung der Einheiten 6.1 - 6.3 der Vorlesung "Einführung in die Theoretische Informatik II".*

Aktuell führt der Artikel lediglich die Komplexitätstheorie ein, betrachtet die Klassen $\mathcal{P}$ und $\mathcal{NP}$. Er wird über das restliche Semester erweitert und aktualisiert werden. Alle Typos und sonstigen Anmerkungen an talten@uni-potsdam.de - 5.06.14

1 Komplexitätstheorie

Die Komplexitätstheorie ist eines der integralen Forschungsgebiete der Theoretischen Informatik. Sie beschäftigt sich mit dem Ressourcenverbrauch von Problemen (sowie deren Algorithmen), klassifiziert Probleme nach unterschiedlichen Schwierigkeitskriterien und formuliert und sucht Zusammenhänge zwischen Komplexitätsklassen. In diesem Artikel betrachten wir ausschließlich entscheidbare Probleme, da semi-entscheidbare Sprachen potentiell unendlich viele Ressourcen benötigen.
Prinzipiell können Komplexitätsanalysen über jede Art von Ressource geführt werden, in diesem Artikel beschränken wir uns auf das Zeit-und Platzverhalten von Problemen.
Die Betrachtung der Komplexität, also den Ressourcenbedarfs, von Problemen ist motiviert durch die einfache Tatsache, dass wir in der Praxis nur begrenzt viele Ressourcen zur Verfügung haben. Vielleicht wird die Wissenschaft irgendwann soweit sein, diese Grenzen zu druchbrechen, doch nach aktuellem Stand besitzten wir maximal soviel Platz und Zeit, wie unser Universum uns zur Verfügung stellt. Und das ist wenig - es gibt Probleme, die bereits für recht kleine Instanzen soviel Zeit erfordern, dass wir leicht 300 oder auch 300 000 Lebenszyklen unseres Universums mit der Lösungsfindung beschäftigt wären. Wer ein wenig über Astronomie Bescheid weiß, ist sich bewusst, dass wir bereits lachhaft riesige Zahlen benutzen, um ein einziges Universum (unser aktuelles) beschreiben zu können. Verglichen mit dem Problem, eine ausgiebige Städtereise optimal zu planen, ist das Universum dann doch etwas zu kurz. Verglichen mit dem Problem, eine schöne Partie Go zu spielen, ist das Universum dann doch etwas zu klein. Wir haben nur schätzungsweise 10^{80} Atome im Universum - wir müssen haushalten, wenn wir boolesche Formeln lösen möchten, die mehr als 260 Variablen haben, denn selbst wenn wir jede mögliche Belegung auf je ein Atom schreiben könnten, würden uns die Atome ausgehen.
Dies muss alles etwas unglaubwürdig klingen - es gibt hunderte Anbieter, die Städtereisen planen, SAT Solver, die Formeln mit hunderttausenden Variablen lösen, und Menschen, die Go spielen. Dies im Sinn, können wir ernsthaft behaupten, dass Komplexitätstheorie von praktischer Relevanz ist? Absolut. Komplexitätstheorie verrät uns unter anderem, dass keines dieser Probleme bislang eine tatsächlich effiziente Lösung besitzt. Wir wissen, dass kleinste Änderungen in der Eingabe die Laufzeit eines Algorithmus von einer Millisekunde auf 400 Sekunden, von

400 Sekunden auf zehn Jahre, von zehn Jahren auf unerahnte Größen anheben können. Das liegt an der Tatsache, dass selbst die besten Algorithmen, die wir kennen um manche Probleme zu lösen von einer Eingabe(länge) zur nächsten explosionsartige Anstiege in ihrer Laufzeit haben. In der Komplexitätstheorie können wir solche Eigenschaften eines Problems analysieren und verallgemeinern. Während wir dank der Berechenbarkeitstheorie erkennen können, welche Programme nicht existieren können (und somit keine Ressourcen darauf verwendet werden sollten, zu versuchen ein solches zu schreiben), können wir dank der Komplexitätstheorie sagen, ob ein Problem überhaupt praktisch fassbar ist.
Die Theorie trifft Aussagen über alle Schwereklassen, auch über die einfachsten und die abstrusesten (Analyse von Algorithmen auf paralellen Rechnern, die mit einer Wahrscheinlichkeit von $1/3$ daneben liegen, ..), doch es ist ersichtlich, weshalb Komplexitätstheorie praktischen Nutzen besitzt und nicht nur ein theoretisches Gespinst ist.
Dieser Artikel gibt eine knappe Einführung in die Komplexitätstheorie und richtet sich vor allem Studenten der Informatik und verwandter Fächer, die neu auf dem Gebiet sind. Wir setzten ein Verständnis über Turinmaschinen, boolesche Logik und mathematische Konzepte, hauptsächlich Notation von Mengen und Funktionen, voraus.
Zunächst müssen wir uns darauf einigen, was Zeit und Platz sind, bevor wir über deren Verbrauch und Verlauf sprechen können. Wir orientieren uns für diese Definitionen an Turingmaschinen - später argumentieren wir, dass es für die zwei Problemklassen, die uns insbesonders interessen, von keiner Relevanz ist, ob unsere Zeit auf Turingmaschinen, Computern oder jeder anderen turingmächtigen sequentiellen Apparatur definiert ist.

Definition 1. *Sei M eine Turingmaschine und w eine beliebige Eingabe über M's Eingabealphabet Σ.*
$t_M(w) = \{n \mid n$ *ist die Anzahl der Konfigurationsübergänge bis M auf w anhält* $\}$
Hinweis: Lass i die Gödelnummer von M sein, dann gilt $t_{M_i}(w) = \Phi_i(w')$ wobei w' die Binärdarstellung von w und Φ_i die Schrittzahlfunktion von M_i ist.

t_M ist also der Zeitbedarf unserer Turingmaschine auf einer spezifischen Eingabe. Während deterministische Turinmaschinen auf jeder Eingabe eine eindeutige Konfigurationsfolge besitzen, können nichtdeterministische Turingmaschinen mit mehreren Konfigurationen weiterarbeiten. NTMs halten entweder mit der ersten akzeptierenden Folge oder mit der letzten nicht akzeptierenden. Aus diesem Verhalten folgt direkt der Funktionswert, welcher entweder die Länge der ersten akzeptierenden Konfigurationsfolge, oder falls eine solche nicht existiert, die der längsten nicht akzeptierenden Konfigurationsfolge ist.
Ähnlich definieren wir den spezifischen Platzbedarf.

Definition 2. *Sei M eine Turingmaschine und w eine beliebige Eingabe über M's Eingabealphabet Σ.*
$t_S(w) = \{n \mid n$ *ist die Anzahl der besuchten Bandzellen bis M auf w anhält* $\}$
Hinweis: Alle Pfade einer NTM teilen sich einen einzigen Speicher. Wurde eine Zelle von einem Pfad besucht, wird sie nicht nochmal gezählt, wenn ein anderer Pfad diese Zelle das erste Mal besucht.

Doch um Komplexitätsanalysen zu führen, sind die Verläufe von Zeit-und Platzbedarf weitaus bedeutender für uns als konkrete Werte auf konkreten Eingaben. Erfahrungsgemäß macht es für viele Zwecke Sinn die Entwicklung der Laufzeit und des Speicherverbrauchs eines Algorith-

mus' oder Problems in Abhängigkeit der Länge der Eingabe zu betrachten. Wir definieren uns nun die nötigen Funktionen, um dies zu ermöglichen.

Definition 3. *Sei M eine Turingmaschine, $n \in \mathbb{N}$ und w eine beliebige Eingabe über M's Eingabealphabet Σ. Wir definieren:*
$T_M(n) = max\{t_M(w) \mid |w| = n\}$

Der Graph von $T_M(n)$ bildet den Zeitverlauf von M ab. Da wir das Maximum für jede Eingabelänge fordern, ist dies der worst-case Verlauf. Wir könnten max durch avg (average) oder min (minimum) ersetzen, um M auf ihre Durchschnittslaufzeit zu untersuchen oder eine best-case Analyse durchzuführen. Verschiedene Anwendungsbereiche der Informatik messen unterschiedlichen Aspekten der Komplexität eines Algorithmus verschiedene Bedeutung bei. So sollte eine Einwegfunktion in der Datensicherheit an keiner Stelle niedrige best-case Werte aufzeigen, während Dienstleister für gewöhnlich keine Programme verwenden wollen, die nur in den meisten aller Fälle schnell sind, aber hin und wieder eine Ewigkeit brauchen, um eine Antwort zu liefern. Im Rahmen dieses Artikels konzentrieren wir uns auf die worst-case Analyse. Es ist oft einfacher darüber zu argumentieren wie viel Zeit und Speicher ein bestimmtes Problem maximal in Anspruch nimmt, bis es gelöst ist, als zu zeigen, dass es für das Problem keinen besseren Algorithmus gibt.
Konvention: Wenn $T_M(n) = f(n)$ ist, sagen wir M ist eine $f(n)$ Turingmaschine. Wir meinen, ihre Laufzeit lässt sich durch $f(n)$ ausdrücken. Dies erspart uns ein wenig Schreibarbeit in manchen Beweisen später.

Definition 4. *Sei M eine Turingmaschine, $n \in \mathbb{N}$ und w eine beliebige Eingabe über M's Eingabealphabet Σ. Wir definieren:*
$S_M(n) = max\{s_M(w) \mid |w| = n\}$

1.1 $\mathcal{O}$-Notation

Als nächstes führen wir einige wichtige Werkzeuge ein, um den Verlauf unterschiedlicher Funktionen in Relation setzen zu können. Dies ist unerlässlich, um Probleme in Schwereklassen (Komplexitätsklassen) einzuordnen, und erlaubt es uns, Schätzungen über die Effizienz von Algorithmen zu geben.
Obwohl dieser Abschnitt $\mathcal{O}$-Notation (gesprochen: O-Notation) benannt ist, führen wir neben $\mathcal{O}$ auch o, Ω und Θ ein (o, Omega, Theta). Wir werden gleich sehen, welche Bedeutung wir jedem Zeichen zumessen.

Definition 5. *Seien $f, g : \mathbb{N} \to \mathcal{R}^+$. Wir sagen f ist eine obere Schranke von g:*

$$g(n) = \mathcal{O}(f(n))$$

g.d.w. $\exists c, wp \in \mathbb{N}$ *sodass* $\forall n \geq wp \in \mathbb{N}$.

$$g(n) \leq c * f(n).$$

Wir schreiben: $g \leq_a c * f$
Hinweis: In den Vorlesungen wird wp (Wendepunkt) n_0 genannt.

f ist eine (asymptotisch) obere Schranke von g, wenn es einen Wendepunkt wp gibt, ab welchem f immer größer (oder gleich) g ist. Das bedeutet, dass die Funktionswerte von f ab einem bestimmten Punkt wp immer schneller (oder gleichschnell) wachsen als die Funktionswerte von g. Wir sagen, f wächst schneller als g. Die Konstante c erlaubt es uns von anderen Konstanten

in f und g zu abstrahieren. c 'unterdrückt' konstante Unterschiede zwischen f und g, da diese irrelevant für das Wachstumsverhalten sind.

Beispiel 6 $g(n) = 3n^3 = \mathcal{O}(n^3) \rightarrow f(n) = n^3 \mid c \geq 3 \; (wp = 0)$

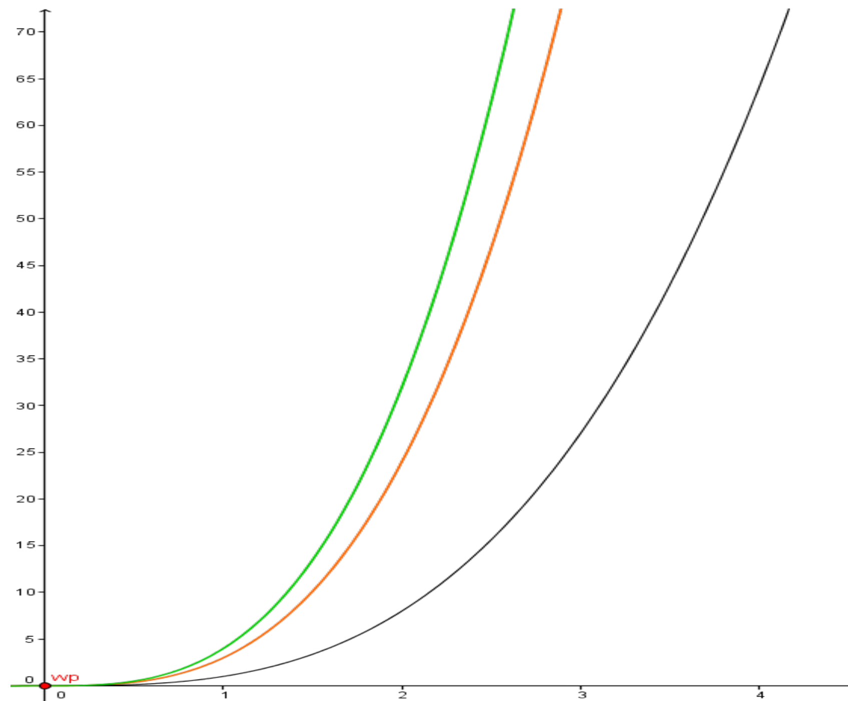


Abbildung 1. $g(n) = 3n^3$ in Orange. $f(n) = n^3$ in Schwarz. $c = 4, c * f(n) = 4 * n^3$ in Grün. Der Wendepunkt ist 0, da $c * f(n)$ bereits ab hier mindestens so schnell wächst wie $g(n)$. Achtung: die Funktionen sind über $\mathcal{R} \rightarrow \mathcal{R}$ visualisiert.

Wie wir an Beispiel 6 erkennen können, ist n^3 eine obere Schranke für $3n^3$. Die mag erst etwas merkwürdig erscheinen, da $3n^3$ schließlich offensichtlich schneller wächst als n^3, doch wie wir bereits erwähnt haben, wollen wir konstante Unterschiede vernachlässigen. Wir wählen c einfach so, dass $c * n^3$ mindestens genauso schnell wächst. Wir haben uns für $c = 4$ entschieden.
Wer noch wenig mit Theoretischer Informatik zu tun hatte, oder generell eher zur angewandten und praktischen Seite lehnt, mag denken, dass konstante Werte einen gewaltigen Unterschied machen. Ob ein Program doppelt so schnell, oder gar zehnmal so schnell läuft wie zuvor kann doch nicht irrelevant sein! Natürlich ist das korrekt, und für die meisten praktisch relevanten Programme ist jede noch so kleine Steigerung der Effiziens wünschenswert - doch dies ist nicht der Fokus der Komplexitätstheorie, die wir in diesem Artikel behandeln. Wir werden Aussagen über ganze Klassen von Problemen treffen, und um dies möglichst einfach zu gestalten, abstrahieren wir von den für uns irrelevanten Details. Konstanten sind ein solches Detail.

Beispiel 7 $g(n) = 3n^3 + 3^n = \mathcal{O}(3^n) \rightarrow f(n) = 3^n \mid c \geq 2 \; (wp = 6)$

Definition 8. *Seien $f, g : \mathbb{N} \rightarrow \mathcal{R}^+$. Wir sagen f ist eine echte obere Schranke von g:*

$$g(n) = o(f(n))$$

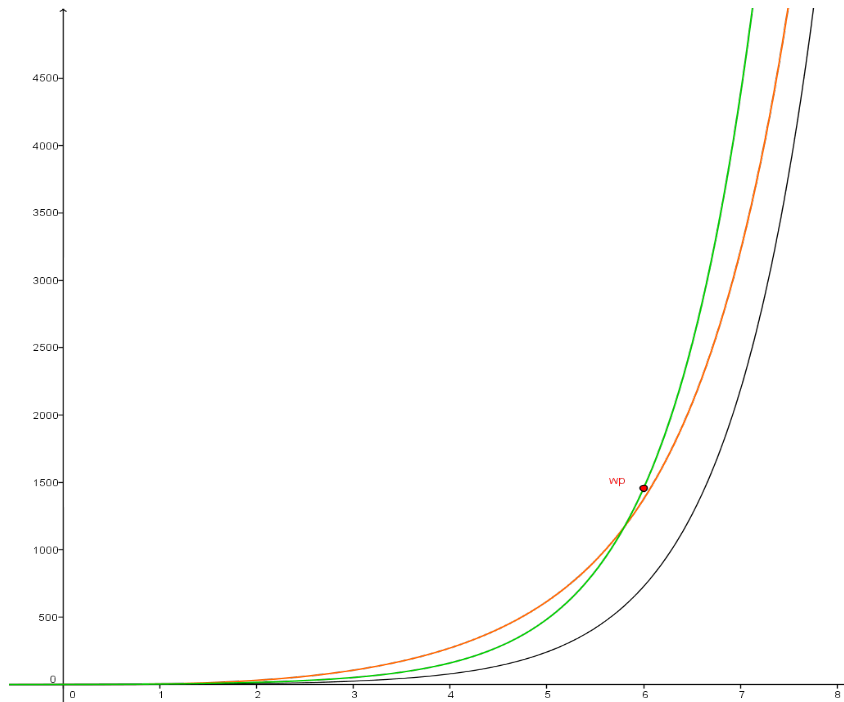


Abbildung 2. $g(n) = 3n^3 + 3^n$ in Orange. $f(n) = 3^n$ in Schwarz. $c = 2, c * f(n) = 2 * 3^n$ in Grün. Der Wendepunkt ist die erste natürliche Zahl, ab der $c * f(n)$ mindestens so schnell wächst wie $g(n)$ (hier $wp = 6$). Achtung: die Funktionen sind über $\mathcal{R} \to \mathcal{R}$ visualisiert.

g.d.w. $\exists c, wp \in \mathbb{N}$ *sodass* $\forall n \geq wp \in \mathbb{N}$.

$$g(n) < c * f(n).$$

Wir schreiben: $g <_a c * f$

Man kann analog zur echten oberen Schranke eine echte untere Schranke definieren, doch wie bereits erwähnt, interessieren uns die exakten unteren Schranken in diesem Artikel weniger. Die einfache untere Schranke genügt für unsere Zwecke. Wir definieren die untere Schranke analog zur oberen.

Definition 9. *Seien* $f, g : \mathbb{N} \to \mathcal{R}^+$. *Wir sagen f ist eine untere Schranke von g:*

$$g(n) = \Omega(f(n))$$

g.d.w. $\exists c, wp \in \mathbb{N}$ *sodass* $\forall n \geq wp \in \mathbb{N}$.

$$g(n) \geq c * f(n).$$

Wir schreiben: $g \geq_a c * f$

Beispiel 10 $g(n) = 3n^3 = \Omega(n^3) \to f(n) = n^3 \mid 0 < c \leq 3$
$g(n) = 3n^3 + 3^n = \Omega(3^n) \to f(n) = 3^n \mid c = 1$
Die Beispielgraphen sind die gleichen wie in den vorherigen zwei Beispielen.

Definition 11. *Seien* $f, g : \mathbb{N} \to \mathcal{R}^+$. *Wir sagen f ist die exakte Schranke von g:*

$$g(n) = \Theta(f(n))$$

g.d.w. $\exists c, c', wp \in \mathbb{N}$ *sodass* $\forall n \geq wp \in \mathbb{N}$.

$$c' * f(n) \leq g(n) \leq c * f(n).$$

Wir schreiben: $c' * f \leq_a g \leq_a c * f$
Hinweis: Man kann leicht zeigen, dass zu jeder Funktion g eine exakte Schranke f existiert: $f(n) = g(n) \Rightarrow g = \mathcal{O}(f) = \Omega(f) = \Theta(f)$.

Beispiel 12 $g(n) = 3n^3 = \Theta(n^3) \rightarrow f(n) = n^3 \mid 0 < c' \leq 3 \leq c$

Wir haben den Begriff asymptotisch erwähnt, als wir die obere Schranke definiert haben. Eine Asymptote ist eine Funktion, die sich einer anderen Funktion im Unendlichen annähert. Wir sprechen von asymptotischen Schranken f, weil, wenn wir nur lang genug warten, f eine Schranke von g bildet. Oder anders ausgedrückt, wir müssen nur endliche lange auf den Wendepunkt warten, bis f unendlich lang die Schranke zu g ist. Jede Schranke, die wir definiert haben, nennt man dementsprechend auch asymptotische Schranke.
Konvention: Wir sagen Turingmaschine M hat Zeitkomplexität $\mathcal{O}(f)$, falls $T_M \in \mathcal{O}(f)$.
Wir sagen Turingmaschine M hat Platzkomplexität $\mathcal{O}(f)$, falls $T_S \in \mathcal{O}(f)$.

1.2 Deterministische Zeitkomplexität

Ab hier werden wir uns auf die Zeitkomplexität konzentrieren, und den Platzbedarf vorerst beiseite lassen. Der Rest dieses Abschnittes, sowie der gesamte Teil 2 des Artikels werden sich ausschließlich mit der Ressource Zeit beschäftigen.

Definition 13. *Sei P eine Sprache. P hat deterministische Zeitkomplexität $\mathcal{O}(f)$ g.d.w. $\exists$ DTM M mit $T_M \in \mathcal{O}(f)$ und $L(M) = P$.*

Statt Sprache hätten wir auch Problem schreiben können, daher der Bezeichner P. Tatsächlich sind die Begriffe Menge, Funktion, Problem und Sprache gleichbedeutend, und wir benutzen sie dementsprechend. Jede Funktion ist lediglich eine Menge ihrer Eingabe-Ausgabe-Paare, jede Sprache ist eine Menge, jede Menge ist eine Sprache, und ein Problem stellen wir als Menge aller Elemente dar, für die das Problem eine Lösung hat. Damit sollte klar sein, dass wir nicht von unterschiedlichen Dingen sprechen, wenn wir ein Problem einmal als Menge, als Funktion oder als Sprache bezeichnen.
Es gibt jede Menge Komplexitätsklassen (unendlich viele, um keine Zahl zu nennen), und die Komplexitätsklassen $\mathcal{P}$ und $\mathcal{NP}$, mit denen wir uns in Kürze näher auseinandersetzen wollen, sind herleitbar.

Definition 14. $TIME(f) = \{P \mid P$ *hat deterministische Zeitkomplexität* $\mathcal{O}(f)\}$

Das heißt, $TIME(f)$ ist die Menge aller Probleme (Sprachen), die von einer DTM in Zeit $\mathcal{O}(f)$ akzeptiert werden. Bevor wir aus dieser allgemeinen Definition die Klasse $\mathcal{P}$ herleiten, kommen wir darauf zurück, weshalb wir den Zeitbedarf auf Turingmaschinen definiert haben, und warum uns dies nicht stört.
Dazu führen wir zwei Beweise: der erste wird zeigen, dass eine Turingmaschine mit beliebig vielen Bändern effizient auf eine äquivalente, einbandige Turingmaschine reduziert werden kann. Dies soll Anlass geben, zu glauben, dass die Auswahl des Maschinenmodells für die Definition der Zeit keine Rolle spielt, solange wir über polynomielle Komplexität sprechen (bei kleinerer, logarithmischer Komplexität sehe die Sache leicht anders aus). Den tatsächlichen Beweis dafür werden wir nicht führen, da dies zu weit gehen würde, doch wir wollen seine Aussage zumindest glaubwürdig machen.

Der zweite Beweis wird demonstrieren, dass man für die Transformation einer Nichtdeterministischen Turingmaschine (NTM) auf eine äquivalente DTM im schlimmsten Fall einen exponentiellen Zeitanstieg in Kauf nehmen muss. Dies ist relevant, da es in der Praxis einen großen Unterschied zwischen Programmen mit polynomieller und exponentieller Laufzeit gibt.

Theorem 15 *Sei f(n) eine Funktion mit $f(n) \geq n$ (monton wachsend). Jede mehrbändrige Turingmaschine mit $T_M = f$ hat eine äquivalente einbandige Turingmaschine M' mit $T_{M'} \in \mathcal{O}(f^2(n))$.*

Beweis: Wir werden den Beweis bewusst grobzügig halten und liefern nur die nötigen Ideen, um die Details rekonstruieren zu können. Zunächst werfen wir einen kurzen Rückblick auf die Umformung einer mehrbändrigen auf eine einbandige Turingmaschine. Dann analysieren wir die Laufzeit dieser Umformung, um festzustellen, dass jeder einzelne Schritt einer mehrbändrigen $f(n)$ Turingmaschine mit einem Band in maximal $\mathcal{O}(f(n))$ Schritten simuliert werden kann. Da eine $f(n)$ Turingmaschine höchstens $f(n)$ Schritte ausführen kann, braucht eine einbandige Turingmaschine höchstens $f(n) * \mathcal{O}(f(n)) = \mathcal{O}(f^2(n))$ Schritte für ihre Simulation.

Eine *einbandige Turingmaschine E* simuliert eine *mehrbändige Turingmaschine K*, indem sie den Inhalt aller Bänder K's auf ihr eigenes Band schreibt. Sie trennt jedes simulierte Band mit einem festgelegten Zeichen, um diese auseinanderhalten zu können, und markiert die Kopfposition eines jeden Bandes. In jedem Schritt von E muss E jedes Band von K und dessen Schreibkopfposition simulieren. Die Übergangsfunktion von K ist in E eincodiert, also weiß E, wie K auf jedem seiner Bänder arbeitet.

Um K zu simulieren, muss E einmal ihr gesamtes Band scannen, um an Hand von K's Übergangsfunktion festzustellen, wie sie die simulierten Bandinhalte und Kopfpositionen verändern muss. Diese Änderungen erfordern einen Rücklauf über E's Band. Um die Laufzeit abzuschätzen, betrachten wir, wie viel Zeit E höchstens benötigt, um über ihr gesamtes Band zu laufen:

Auf E's Band stehen k Bandinhalte, wenn K k Bänder hat. Jeder dieser Bandinhalte kann maximal $f(n)$ lang sein (da K nur maximal so viele Bandzellen aufsuchen kann, wie sie Zeit hat). Das heißt, E's Band ist höchstens ein Vielfaches von $f(n)$ lang, also in $\mathcal{O}(f(n))$. Desweiteren muss E nur zweimal pro Schritt über ihr komplettes Band laufen, und immer über einen Teil davon, wenn eines der simulierten Bänder eine neue Bandzelle aufsucht. E's Laufzeit pro Schritt bleibt also weiterhin in $\mathcal{O}(f(n))$. Um K vollständig zu simulieren, muss E höchstens $f(n)$ Schritte ausführen. E simuliert K also in $f(n) * \mathcal{O}(f(n)) = \mathcal{O}(f^2(n))$ Schritten. □

Aus diesem Beweis folgt, dass man von mehrbändrigen auf einbandige TMs lediglich quadratisch, also polynomiellen Zeitzuwachs hat. Tatsächlich kann man zeigen, dass jedes deterministische, turingmächtige Berechenbarkeitsmodell mit maximal polynomiellen Zeitzuwachs in DTMs umgewandelt werden kann. Insbesondere gilt dies auch für den Computer. Der Beweis würde an dieser Stelle zu weit hinausgehen, für die Idee siehe [2]. Die Intuition ist, dass man jedes solches Maschinenmodell mit einer genügend großen Zahl von Bändern relativ leicht simulieren kann.

Theorem 16 *Sei f(n) eine Funktion mit $f(n) \geq n$ (monton wachsend). Jede nichtdeterministische Turingmaschine mit $T_M = f$ hat eine äquivalente deterministische, einbandige Turingmaschine M' mit $T_{M'} \in 2^{\mathcal{O}(f(n))}$.*

Stimmt diese Aussage, so bedeutet dies, dass ein nichtdeterministischer Algorithmus im schlimmsten Fall auf jedem turingmächtigen, deterministische Berechenbarkeitsmodel exponentiellen Zeitanstieg benötigt. Es ist schnell ersichtlich, dass der Unterschied zwischen polynomiellem und exponentiellem Zeitverbrauch praktisch relevant ist. Betrachten wir einen Algorithmus, der in n^2 arbeitet, und einen in 2^n. Ersterer ist ein kleines Polynom, zweiterer eine sehr kleine Exponentialfunktion. Nehmen wir eine geringe Eingabelänge von 100. Der polynomielle Algorithmus würde $100^2 = 10000$ Schritte benötigen, was zwar nach viel aussieht, doch im Grunde wenig ist, wenn man sich vor Augen führt, dass ein 1GHz Prozessor 1000000000 Takte pro Sekunde ausführt. Nun ist nicht jeder Schritt eines Algorithmus gleichbedeutend mit Takt eines Prozessors, doch es ist offensichtlich, dass der polynomielle Algorithmus sehr schnell fertig wäre. Betrachten wir 2^{100} und plötzlich erhalten wir eine astronomische Schrittzahl mit 31 Dezimalstellen. Selbst die besten Prozessoren würden eine Weile daran zu arbeiten haben. Als Gedankenexperiment: Würde jeder Takt eines 1GHz Prozessors tatsächlich eine Milliarde Schritte des Algorithmus ausführen können, bräuchten wir bei einem Takt pro Sekunde in etwa soviele Sekunden wie es Sandkörner auf unserem Planeten gibt.

Beweis: Die Beweisidee ist simpel. Wir stellen uns die Abarbeitung der NTM als ein Baum vor. Jeder Knoten repräsentiert einen Entscheidungspunkt, jeder Zweig eine Entscheidung. Wir summieren die Zeit auf, die eine DTM bräuchte, um jeden Pfad im Baum sequentiell zu simulieren.

Sei *N eine $f(n)$ NTM* - das bedeutet jeder Pfad in N's Abarbeitungsbaum ist höchstens $f(n)$ lang. Sei *D eine DTM*, wobei es irrelevant ist, auf wie vielen Bändern D arbeitet, wie wir am Ende sehen werden.

Lass *b die maximale Anzahl möglicher Entscheidungen* von N sein, dann hat jeder Knoten in N's Arbeitungsbaum maximal b Kinder. Da jeder Pfad höchstens $f(n)$ lang ist, hat der Baum maximal $b^{f(n)}$ Blätter. Wir wissen, dass ein Baum weniger als doppelt so viele Knoten hat, wie dessen maximale Anzahl an Blättern. Wir besuchen also $\mathcal{O}(b^{f(n)})$ Knoten während unserer Simulation. Idealerweise würden wir jeden Knoten nur ein einziges Mal besuchen, doch selbst wenn wir von einer ineffizienteren Variante ausgehen, in welcher wir nach der Bearbeitung eines Pfades wieder von der Wurzel des Baumes anfangen, führt die Simulation nur bis zu $\mathcal{O}(f(n) * b^{f(n)})$ Schritte aus. Es gilt: $\mathcal{O}(f(n) * b^{f(n)}) = 2^{\mathcal{O}(f(n))}$

Es ist irrelevant wie viele Bänder D hatte, da wir sie unter quadratischem Anstieg in eine einbandige Maschine umwandeln können. Das Quadrat ändert die Gültigkeit unserer Herleitung nicht.

Herleitung ($f(n)$ wurde aus Lesbarkeitsgründen durch n ersetzt):

$$\mathcal{O}(n * b^n) = 2^{\mathcal{O}(n)}$$
$$log_2(\mathcal{O}(n * b^n)) = \mathcal{O}(n)$$

$$\mathcal{O}(log_2(n * b^n)) = \mathcal{O}\left(\frac{log(n*b^n)}{log(2)}\right) = \mathcal{O}(log(n * b^n)) = \mathcal{O}(log(n) + n * log(b)) = \mathcal{O}(n)$$

□

2 $\mathcal{P}$ - $\mathcal{NP}$

Definition 17. $\mathcal{P} = \bigcup_k TIME(n^k)$

Dies macht $\mathcal{P}$ zur Klasse aller Probleme, die deterministisch in polynomieller Zeit gelöst werden können. Die meisten Programme, die in der Praxis geschrieben und genutzt werden, liegen innerhalb dieser Klasse. Wir haben bereits vor dem Beweis von Theorem 16 motiviert, dass ein unübersehbarer Unterschied zwischen polynomiellen und exponentiellen Problemen besteht. Es gibt jedoch viele exponentielle Probleme, die von großer praktischer Relevanz sind. Boolesche Formeln zu lösen, Pfade und Wege mit bestimmten Eigenschaften finden, Elemente unterschiedlicher Größe auf eine Menge von Behältern verteilen und viele weitere Problemstellungen müssen angegangen werden. Wir müssen rausfinden können, ob Schaltkreise korrekt funktionieren, und dafür müssen wir Formeln lösen können. Wir wollen optimale Wege fahren, wenn wir unser GPS nach einer Route fragen. Wir wollen Pakete möglichst effizient versenden können, und dafür müssen wir diese Pakete packen können. Jedes dieser Probleme hat optimale Lösungen, die berechnet werden können, doch sie alle liegen in der schweren Klasse $\mathcal{NP}$, von der die meisten Informatikstudenten bereits gehört haben dürften. Wir wollen diese Klasse von hieran definieren und rausfinden, was diese simplen Probleme so schwer macht, dass wir bis hierher behauptet haben, dass es keine bekannten, effizienten Algorithmen gibt, um sie zu lösen.

Definition 18. *Sei P eine Sprache. P hat nichtdeterministische Zeitkomplexität $\mathcal{O}(f)$ g.d.w. $\exists$ NTM M mit $T_M \in \mathcal{O}(f)$ und $L(M) = P$.*

Definition 19. $NTIME(f) = \{P \mid P \text{ hat nichtdeterministische Zeitkomplexität } \mathcal{O}(f)\}$

Definition 20. $\mathcal{NP} = \bigcup_k NTIME(n^k)$

$\mathcal{NP}$ ist also die Klasse all derer Probleme, die mit einem nichtdeterministischen Berechenbarkeitsmodel in polynomieller Zeit gelöst werden können. Bevor wir den Zusammenhang zwischen $\mathcal{P}$ und $\mathcal{NP}$ weiter ausführen, führen wir eine alternative Möglichkeit ein, um festzustellen, ob ein Problem in $\mathcal{NP}$ liegt.

Definition 21. *Ein Verifizierer V für eine Sprache L ist ein Algorithmus, sodass gilt*
$L = \{w \mid V \text{ akzeptiert } < w, c > \text{ für ein Wort c } \}$
Wir sagen, V ist ein Verifizierer für L.
Hinweis: c wird Beweis oder Zertifikat für w genannt.

In anderen Worten, ein Verifizierer ist in der Lage an Hand eines Zertifikates c zu entscheiden, ob w in der Sprache liegt oder nicht. Ist die Laufzeit des Verifizierers polynomiell im Verhältnis zur Eingabegröße $|w|$, wissen wir, dass die Sprache, die er entscheidet, in $\mathcal{NP}$ liegt. Dies beruht auf der einfachen Tatsache, dass jede Lösung eines $\mathcal{NP}$ Problems höchstens polynomiell lang ist. Wenn wir also eine Lösung c für ein beliebiges $\mathcal{NP}$ Problem bekommen, sind wir in der Lage in polynomiell vielen Schritten nachzuvollziehen (zu verifizieren), ob c tatsächlich eine Lösung ist. Wir werden nach dem folgenden Beispiel den Beweis dazu führen.

Beispiel 22 Wir betrachten zur Veranschaulichung das $\mathcal{NP}$-Problem $HAMPATH$. Es besteht darin, zu bestimmen, ob ein gerichteter Graph einen hamilton'schen Pfad besitzt. Ein Pfad ist

hamilton'sch, wenn er jeden Knoten des Graphen genau einmal durchläuft.
Da Pfad, Weg, Kantenfolge etc. in verschiedener Literatur teils leicht unterschiedlich definiert werden, bestimmen wir unsere Definition von Pfad. Ein Pfad ist eine Folge von Knoten, in welcher je aufeinanderfolgende Knoten durch eine Kante verbunden sind, und in der kein Knoten mehrfach auftritt. (Das heißt, der Pfad schneidet sich nicht.)
$HAMPATH = \{< G, s, t > | G$ ist ein gerichteter Graph mit einem hamilton'schen Pfad von Knoten s nach Knoten $t\}$
Wir kreieren einen *Verifizierer* V, der für eine Eingabe $\langle < G, s, t >, [n_1, .., n_k] \rangle$ in polynomieller Zeit entscheidet, ob $< G, s, t > \in HAMPATH$. Dabei ist $[n_1, .., n_k]$ der angeblich hamilton'sche Pfad. Wir prüfen lediglich, ob dies stimmt.
V := " auf Eingabe $\langle < G, s, t >, [n_1, .., n_k] \rangle$ tue:

1. Prüfe, ob die Eingabe syntaktisch korrekt ist. Wenn nicht, REJECT.
2. Prüfe, ob $\forall n_i$ Knoten aus G sind. Wenn nicht, REJECT.
3. Prüfe, ob jeder Knoten nur einmal vorkommt. Wenn nicht, REJECT.
4. Prüfe, ob es für jeden Knoten x in G ein n_i gibt, sodass $x = n_i$. Wenn nicht, REJECT.
5. Prüfe, ob $[n_1, .., n_k]$ ein Pfad ist. Das heißt, ob für jedes (n_i, n_{i+1}) eine Kante (x, y) in G existiert, mit $n_i = x, n_{i+1} = y$. Wenn ja, ACCEPT, sonst REJECT. "

In den Folien nutzen wir Verifizierer im Zusammenhang mit Orakeln. Das Orakel gibt uns dabei das Zertifikat in einem ersten Schritt, während wir hier voraussetzen, dass uns das Zertifikat von irgendwo her gegeben wird. Es ist das gleiche Prinzip.
Der Algorithmus muss prüfen, ob unser Zertifikat ein Pfad in G ist, und ob dieser hamilton'sch ist. Die ersten drei Schritte sind einfache Ausschlussfälle - alle Knoten im Pfad müssen aus G kommen, und es darf kein Knoten mehrfach vorkommen, da es sonst kein Pfad ist. Im vierten Schritt prüfen wir, ob der Pfad hamilton'sch ist, da in einem solchen jeder Knoten von G passiert wird. Im letzten Schritt versichern wir uns, dass der Pfad tatsächlich ein Pfad ist, indem wir prüfen, ob alle nötigen Kanten in G vorhanden sind. Der Algorithmus ist also korrekt, nun betrachten wir dessen Laufzeit:
Schritt 1, 2 und 4 erfordern jeweils nur einen Scan über die Eingabe des Verifzierers, sind also in linearer Zeit machbar. In Schritt 3 merkt sich der Algorithmus einfach jeden Knoten, den er bereits gesehen hat, und REJECT, falls er den gleichen nocheinmal ließt. Das erfordert maximal k Scans des Zertifikats, bleibt also linear. Schritt 5 führt $k/2$ Vergleiche aus, womit der gesamte Algorithmus in polynomieller Zeit läuft.
Da V ein polynomer, korrekter Verifizierer für $HAMPATH$ ist, gilt $HAMPATH \in \mathcal{NP}$.

Wir haben zwar bereits verraten, dass $\mathcal{NP}$ nicht nur die Klasse der Sprachen ist, die durch NTMs in polynomieller Zeit akzeptiert werden, sondern auch die der Sprachen, die einen polynomiellen Verifizierer haben, doch den Beweis haben wir im Austausch für das obige Beispiel aufgeschoben. Diesen wollen wir nun nachholen.

Theorem 23 *Eine Sprache P liegt in $\mathcal{NP}$ g.d.w. ein polynomieller Verifizierer V für P existiert.*

Beweis: Nach unserer Definition von $\mathcal{NP}$ wissen wir bereits, dass jede Sprache in $\mathcal{NP}$ von einer NTM in polynomieller Zeit akzeptiert wird. Wir müssen also zum Einen zeigen, dass wir aus einer polynomiellen NTM einen polynomiellen Verifizierer der gleichen Sprache machen können, und zum anderen, wie wir einen polynomiellen Verifizierer einer Sprache in eine polynomielle NTM umwandeln können. Wir beginnen damit, aus einer NTM einen

Verifizierer zu konstruieren:
Sei *N unsere polynomielle NTM* mit $L(N) = P$. Wir zeigen, wie wir mit N einen *polynomiellen Verifizierer V* für P bauen können.
$V :=$ " auf Eingabe $< w, c >$ tue:

1. Simuliere N auf w, orientiere dich bei jeder nichtdeterministischen Entscheidung an c.
2. Falls N am Ende der Simulation akzeptiert, ACCEPT. Sonst, REJECT. "

Die Simulation von N ist polynomiell, da jede Entscheidung durch c vorgegeben, und der totale Abarbeitungspfad maximal polynomiell viele Schritte lang ist (da N P in polynomieller Zeit löst). An Hand von N's Übergangsfunktion überprüfen wir in jedem Schritt der Simulation, ob diese korrekt arbeitet. Wenn die Simulation korrekt war und am Ende akzeptiert, akzeptiert auch V, andernfalls lehnt V ab. Dies beweist die eine Richtung unserer Aussage. Als nächstes konstruieren wir die NTM aus dem Verifizierer.
Sei V unser polynomieller Verifizierer für P. Es existiert eine TM V' für V, welche V für eine Konstante k in n^k Schritten ausführt (da V in polynomieller Zeit arbeitet). Wir zeigen, wie wir mit V eine polynomielle NTM N mit $L(N) = P$ bauen können.
$N =$ " auf Eingabe w der Länge n tue:

1. Wähle nichtdeterministisch ein Wort c mit maximaler Länge $n^k - |w|$.
2. Führe V' auf $< w, c >$ aus.
3. Akzeptiert V', ACCEPT. Sonst, REJECT. "

Da V' in spätestens n^k Schritten terminiert, wobei n die Länge seiner Eingabe und k eine Konstante ist, kann die Eingabe selbst nicht länger als n^k sein (V' hätte dann gerade genug Zeit, um seine gesamte Eingabe zu lesen). Das bedeutet, dass das Zertifikat c, falls es ein solches gibt, ebenfalls nicht länger sein kann. Wir können also nichtdeterministisch jedes c mit dieser Maximallänge durch unsere NTM auswählen, und für jedes c V' aufrufen. Wenn ein richtiges Zertifikat gefunden wurde, akzeptiert N, wenn keines dabei war und w dementsprechend nicht Element von P ist, lehnt N ab. N arbeitet korrekt und in polynomieller Zeit. Damit haben wir die zweite Richtung unserer Aussage bewiesen. □

Bislang haben wir nur gesagt, dass die Lösung eines $\mathcal{NP}$ Problems im schlimmsten Fall exponentiell mehr Zeit erfordert, wenn man es auf einer deterministischen Turingmaschine löst. Warum beschränken wir uns dabei auf diesen Härtefall? Wieso sagen wir nicht ganz allgemein, dass es in jedem Fall so ist? Die Antwort ist eines der populärsten ungelösten Probleme unseres derzeitigen Wissenschaftsstandes:

$$\mathcal{P} \stackrel{?}{=} \mathcal{NP}$$

Wir wissen es nicht.
In der gesamten (kurzen) Geschichte der Informatik als Wissenschaft, ist es noch niemandem gelungen, die Gleichheit oder Ungleichheit der beiden Klassen $\mathcal{P}$ und $\mathcal{NP}$ zu beweisen. Wir haben mehrere Probleme genannt, von denen wir behauptet haben, dass sie schwer sind, weil ihre Lösung exponentiell viel Zeit in Anspruch nimmt. Doch das war nur die praktische Schwere - theoretisch betrachtet, können wir nicht garantieren, dass diese Probleme tatsächlich so viel Zeit in Anspruch nehmen. Es ist nur noch niemanden gelungen, einen Algorithmus zu finden, der sie effizient löst. Gleichzeitig ist es auch noch niemandem gelungen, zu belegen, dass man sie nicht effizient lösen kann.
Indizien sprechen jedoch, bedauernswerterweise, gegen die Gleichheit der beiden Klassen. Es

gibt soviele schwere Probleme in $\mathcal{NP}$, für die noch niemand eine effiziente Lösung gefunden hat, dass die Chancen schlecht aussehen, dass dies in Zukunft jemand schafft. Doch dies ist nicht alles, es wurde bereits bewiesen, dass eine ganze Reihe von Beweisverfahren nicht genutzt werden können, um die Gleichheit, bzw. Ungleichheit zu beweisen. So kann man mit Hilfe sogenannter Orakelturingmaschinen zeigen, dass kein Beweis, der sich im Kern auf ein Diagonalisierungsverfahren oder eine Simulation von Maschinen stützt diese Frage lösen wird. Siehe Kapitel 9.2 in [1].
Wir müssen uns also vorerst damit abfinden, dass Probleme, die in $\mathcal{NP}$ liegen tendentiell schwer zu lösen sind, während Probleme in $\mathcal{P}$ tendenziell leicht zu lösen sind. Es gibt für beide Klassen Ausnahmen, wenn wir uns auf praxisbedingte Anwendungsfälle beschränken. So gibt es SAT-Solver, die boolesche Formeln mit hunderttausend Variablen und mehr auswerten können, und polynomielle Algorithmen, die für große, relevante Eingaben ebenfalls länger brauchen als praktikabel.
Wie kann man also die $\mathcal{P} - \mathcal{NP}$ Frage angehen? Ob man zeigen möchte, dass die Gleichheit gilt, oder eben die Ungleichheit, in beiden Fällen macht es Sinn die schwersten Probleme in $\mathcal{NP}$ zu betrachten. Es ist offensichtlich, dass jedes $\mathcal{P}$ Problem in $\mathcal{NP}$ liegt, also verbleibt nur noch zu zeigen, dass ein $\mathcal{NP}$ Problem existiert, welches nicht in $\mathcal{P}$ enthalten ist. Alternativ, wenn man einen Weg findet das schwerste Problem in $\mathcal{NP}$ in polynomieller Zeit zu lösen, dann sollte auf diesem Wege alle übrigen Probleme ebenfalls lösen können. Wir formalisieren diese Intuition.

Definition 24. *Seien L, L' Sprachen. L' ist auf L reduzierbar, falls eine totale, surjektive, berechenbare Funktion $f : L' \to L$ existiert, welche Elemente aus L' in Elemente aus L abbildet. Wir schreiben: $L' \leq L$.*

In anderen Worten, ein Element x ist Element von L' g.d.w. $f(x)$ ein Element von L ist. Reduzierbarkeit ist ein praktisches und mächtiges Beweismittel, um Aussagen über die Komplexität von Problemen und Algorithmen zu treffen. Vor allem wie schwer Probleme in Relation zu anderen Problemen sind. Uns interessieren insbesondere solche Reduktionen, welche die Komplexitätsklasse nicht verändern, da wir so durch Reduktion festellen können, ob zwei Probleme in der gleichen Klasse liegen. Für $\mathcal{NP}$ bedeutet dies, dass unsere Reduktionen in polynomieller Zeit ausführbar sein müssen.

Definition 25. *Seien L, L' Sprachen. L' ist auf L polynomiell reduzierbar, falls eine totale, surjektive, berechenbare Funktion $f : L' \to L$ existiert, welche Elemente aus L' in polynomieller Zeit in Elemente aus L abbildet. Wir schreiben: $L' \leq_P L$.*
Hinweis: In den Folien ist die Bedingung formuliert als $L' = f^{-1}(L)$. Dies ist gleichtbedeutend mit unserer Definition.

Wie wir sehen werden, existieren Probleme in $\mathcal{NP}$, auf die alle anderen Probleme in $\mathcal{NP}$ reduziert werden können. Wir können also die Lösung aller $\mathcal{NP}$ Probleme darauf zurückführen, wie wir diese schwersten Probleme lösen.

Definition 26. *Sei L eine Sprache. Wir nennen L $\mathcal{NP}$-hart (oder $\mathcal{NP}$-schwer), wenn jede Sprache L' in $\mathcal{NP}$ auf L reduzierbar ist.*
Hinweis: $\mathcal{NP}$-hard im Englischen.

$\mathcal{NP}$-harte Probleme sind schwerer als alle $\mathcal{NP}$ Probleme - sie müssen selbst nicht in $\mathcal{NP}$ vorkommen! Sie könnten auch aus einer schwereren Klasse stammen (und davon haben wir, wie bereits angedeutet, sehr viele).

Definition 27. *Sei L eine Sprache in $\mathcal{NP}$. Wir nennen L $\mathcal{NP}$-vollständig, wenn jede Sprache L' in $\mathcal{NP}$ auf L reduzierbar ist.*
Hinweis: $\mathcal{NP}$-complete im Englischen.

Aus der Definition für $\mathcal{NP}$-Vollständigkeit folgt direkt, dass $\mathcal{P} = \mathcal{NP}$, falls es jemand schafft, einen deterministischen, polynomiellen, berechenbaren Algorithmus für ein $\mathcal{NP}$-vollständiges Problem zu beschreiben.
Von hieran wollen wir uns für den Rest des Kapitels mit $\mathcal{NP}$-vollständigen Problemen beschäftigen. Wir werden für ausgewählte Probleme per Reduktion auf ein bereits bekanntes $\mathcal{NP}$-vollständiges Problem beweisen, dass sie ebenfalls $\mathcal{NP}$-vollständigsind. Zuvor müssen wir jedoch ein solches kennenlernen.

Definition 28.

$$SAT = \{k_1, .., k_m \mid k_i \text{ Klausel über } x_1, ..x_n \wedge \\ (\exists a_1, .., a_n \in \{0,1\}. \forall j \leq m. a_1, .., a_n \text{ erfüllt } k_j)\}$$

Eine alternative Definition, die sich leichter ließt, ist Folgende:

$$SAT = \{\varphi \mid \varphi \text{ ist eine erfüllbare Formel in KNF}\}$$

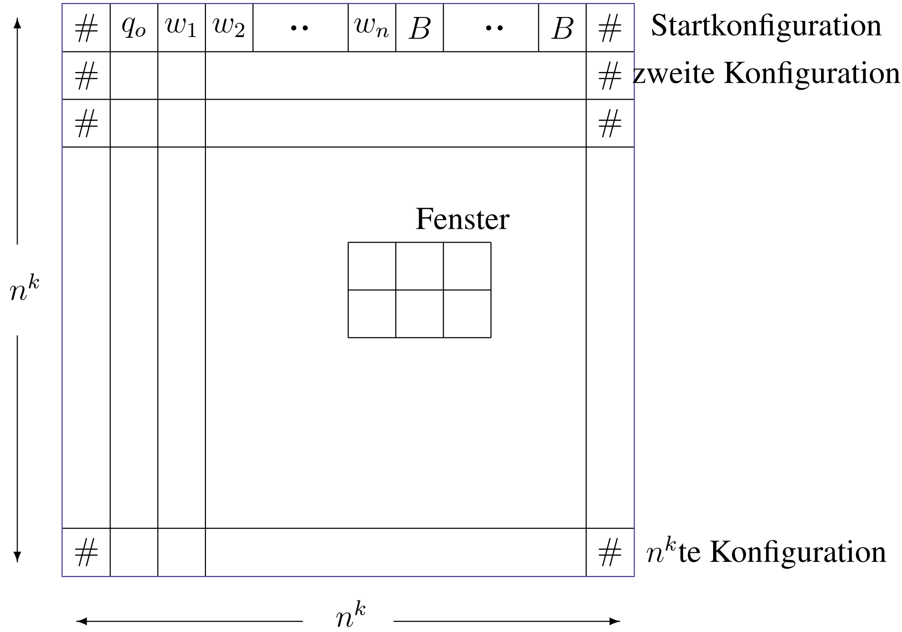
Theorem 29 (COOK-LEVIN-THEOREM) *SAT ist NP-vollständig.*

Der zugehörige Beweis mag zunächst recht aufwendig erscheinen, doch lohnt es sich die Zeit zu nehmen ihn nachzuvollziehen.
Beweise auf $\mathcal{NP}$-*Vollständigkeit* über polynomielle Reduktion sind nur dann aussagekräftig, wenn man bereits ein $\mathcal{NP}$-*vollständiges* Problem kennt. Dafür muss man dieses Problem jedoch ohne Hilfe von Reduktion als $\mathcal{NP}$-*vollständig* bewiesen haben. SAT bietet sich an, da wir intuitiv einsehen, dass alles, was ein Rechner kann, mit logischen Schaltungen funktioniert. Logische Schaltungen sind nichts weiter als boolesche Formeln, und SAT ist genau die Menge aller erfüllbaren Formeln, also gleichzeitig auch aller terminierender Programme und total berechenbarer Funktionen. Wenn es also ein Problem in $\mathcal{NP}$ gibt, sodass wir jedes andere Problem in $\mathcal{NP}$ darauf reduzieren können (d.h. entsprechende total berechenbare Funktionen finden), dann ist es am wahrscheinlichsten SAT.

Beweis: Zunächst erklären wir die Beweisidee:

Um zu beweisen, dass SAT $\mathcal{NP}$-vollständigist, müssen wir zwei Dinge zeigen: SAT liegt in $\mathcal{NP}$; jedes Problem in $\mathcal{NP}$ ist auf SAT reduzierbar.
Wir zeigen, dass $SAT \in \mathcal{NP}$. Wir wissen, dass die Belegung einer Formel im Verhältnis zu ihrer Länge in linearer Zeit ausgewertet werden kann (vgl. Foliensatz 6.2). Unsere NTM, die SAT akzeptiert, wählt nichtdeterministisch jede mögliche Variablenbelegung ihrer Eingabeformel aus und prüft ob eine davon die Formel wahr macht. Somit liegt SAT in $\mathcal{NP}$.
Nun zeigen wir, dass jede Sprache in $\mathcal{NP}$ polynomiell auf SAT reduzierbar ist: Sei $P \in \mathcal{NP}$ ein beliebiges Problem. Lass N_P *die NTM* sein, welche P akzeptiert. N_P arbeitet in $n^k - 2$ Schritten (die -2 sind ein Detail, welches wir großzügig ignorieren können - wir schreiben es nur der Exaktheit halber mit auf), wobei n wie gewohnt für die Eingabelänge und k für eine Konstante steht. Für den Beweis führen wir folgendes Konzept ein: Ein *Tableau für N auf w* ist eine $n^k \times n^k$ Tabelle, deren Zeilen je eine Konfiguration von N auf w

darstellen.



Um uns den Beweis zu erleichtern, nehmen wir an, dass jede Reihe mit einer # beginnt und endet (das sind die -2 von eben), wobei # nicht in N_P selbst vorkommen darf. Sollte es vorkommen, benennen wir es einfach um. Die erste Zeile des Tableaus stellt immer die Startkonfiguration von N auf w dar, und jede weitere Zeile des Tableaus folgt direkt aus ihrer Vorgängerzeile durch die Konfigurationsbeschreibung von N. Wir nennen ein Tableau akzeptierend, wenn eine seiner Zeilen eine akzeptierende Konfiguration darstellt.
Jedes akzeptierende Tableau für N auf w steht für einen akzeptierenden Abarbeitungspfad von N auf w. Die Frage, ob N mit w akzeptiert reduziert sich also auf die Frage, ob ein akzeptierendes Tableau von N auf w existiert. Wir werden eine Formel φ konstruieren, die genau dann erfüllbar ist, wenn ein akzeptierendes Tableau existiert, und unerfüllbar, wenn es keines gibt.
Sei f unsere Reduktion von P auf SAT. f produziert auf Eingabe w die entsprechende Formel φ. Seien Q und Γ die Zustandsmenge und das Bandalphabet von N_P. Sei $C = Q \cup \Gamma \cup \{\#\}$. Für jedes $i, j \in [1, .., n^k]$ und jedes $s \in C$ haben wir eine Variable $x_{i,j,s}$.
Für jedes $i, j \in [1, .., n^k]$ besitzt unser Tableau eine *Zelle*, in Reihe i, Spalte j. Wir bezeichnen sie mit $cell[i, j]$ und sagen, sie enthält s, wenn $x_{i,j,s} = 1$.
Unsere Formel φ besteht aus vier Teilformeln: $\varphi_{cell} \wedge \varphi_{start} \wedge \varphi_{accept} \wedge \varphi_{move}$. Die erste Teilformel erzwingt, dass in jeder Zelle genau ein Symbol s steht. Kein Tableau, in dem mehrere Symbole in einer Zelle vorkommen, kann ein akzeptierendes Tableau sein (da es die Konfigurationssyntax verletzt).

$$\varphi_{cell} = \bigwedge_{1 \leq i,j \leq n^k} \left[\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{s \neq t \in C} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right]$$

Das erste große AND bedeutet, dass jede Zelle betroffen ist. Das erste OR sagt, dass mindestens ein Symbol in der aktuellen Zelle steht. Das zweite große AND erzwingt, dass nicht

mehr als ein Symbol in der aktuellen Zelle steht. Somit haben wir den ersten Teil unserer Formel φ beschrieben.
Die zweite Teilformel φ_{start} stellt sicher, dass die erste Zeile des Tableaus die Startkonfiguration von N_P auf w enhält. Wie diese aussieht, ist durch N_P vorgegeben. Sie besitzt also die Form:

$$\begin{aligned}\varphi_{start} =& x_{1,1,\#} \wedge x_{1,2,q_0} \wedge \\ & x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge .. \wedge x_{1,n+2,w_n} \wedge \\ & x_{1,n+3,B} \wedge .. \wedge x_{1,n^k-1,B} \wedge x_{1,n^k,\#}\end{aligned}$$

Wobei q_0 für den Startzustand von N_P steht, und B für das Blanksymbol. Damit haben φ_{start} beschrieben.
Die dritte Teilformel φ_{accept} garantiert, dass wir in irgendeiner Zelle q_{accept} stehen haben, wobei q_{accept} der akzeptierende Zustand von N_P ist. Da jede Konfiguration unseres Tableaus aus ihrer Vorgängerkonfiguration folgt, stellt dies sicher, dass wir ein akzeptierendes Tableau beschreiben.

$$\varphi_{accept} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_{accept}}$$

Nun schauen wir uns an, wie wir sicherstellen, dass jede Zeile (d.h. jede Konfiguration) korrekt aus ihrer Vorgängerzeile folgt. Dazu betrachten wir jedes 2×3 Fenster unseres Tableaus, und bestimmen, ob es legal (korrekt) ist. Wenn jedes 2×3 Fenster des Tableaus legal ist, wissen wir, dass das gesamte Tableau legal ist, d.h. jede Konfiguration durch die Konfigurationsbescheibung von N_P aus ihrem Vorgänger folgt.
Wir nennen ein Fenster legal, wenn es in einem richtigen Tableau auftauchen könnte. Oder anders gesagt, ein legales Fenster verletzt keine Regeln der Konfigurationsbeschreibung. Wir verdeutlichen dies an einem kleinen Beispiel:
Sei $\delta(q_1, e) = \{(q_1, g, L), (q_1, h, R)\}, \delta(q_2, g) = \{(q_2, e, L)\}$ unsere Übergangsfunktion. Die folgenden (in Abb. 3) sind legale Fenster: Das erste Fenster ist legal, da q_1 auf e die-

q_1	e	h
h	g	h

e	g	h
e	g	q_1

h	e	g
h	e	g

e	h	q_1
e	h	h

Abbildung 3. Beispiele legaler Fenster.

ses in ein g umwandelt und nach Links läuft. Links von q_1 hätte ein h stehen können, das Fenster verletzt also die Konfigurationsbeschreibung nicht und ist dementsprechend legal. Im zweiten Fenster hätte rechts neben h q_1 stehen und durch eine Aktion nach Links laufen können, weshalb es in der unteren Zeile erscheint. h wäre nach Rechts gerutscht. Im dritten Fenster ändert sich nichts von der einen zur nächsten Zeile - dieser Fall tritt immer dann ein, wenn keines der Randsymbole oben neben einem Zustand steht, ist also legal. Im vierten Fenster könnte neben q_1 ein e gestanden haben, welches durch ein h ersetzt wurde und nach

Links gerutscht ist.
Nicht legale Fenster sind unter anderem 4: Das erste Fenster ist nicht legal, da das veränderte

e	h	e
e	g	e

q_1	e	q_2
q_1	q_2	e

h	q_2	g
q_2	h	h

Abbildung 4. Beispiele nicht legaler Fenster.

Symbol nicht neben einem Zustand lag. Das zweite Fenster beinhaltet mehr als einen Zustand (eine Konfiguration hat nur einen Zustand) und im dritten Fenster ersetzt q_2 g durch h anstatt durch e.
Wir werden nicht in die technischen Details eintauchen, wie man formal beschreibt, dass ein Fenster legal ist, da wir für den Beweis nur die Idee benötigen. Als kleine Motivation für den Interessierten Leser: Wir können aus den gegeben Regeln von N_P statisch für jede Übergangsbeschreibung eine Menge an legalen Fenstern erzeugen, die durch den Übergang ermöglicht werden. Wenn wir alle so erzeugten legalen Fenster aller Übergangen zusammennehmen, müssen wir im Tableau nur noch abgleichen, ob die vorkommenen Fenster in unserer Menge legaler Fenster enthalten sind oder nicht.
Wenn jedes Fenster unseres Tableaus legal ist, folgt jede Zeile des Tableaus aus ihrem Vorgänger. Der Beweis dieser Intuition ist einfach - wir wissen, dass die erste Zeile unseres Tableaus korrekt ist, da es die durch q_{start} festgelegte Startkonfiguration ist. Jedes Bandsymbol einer Zeile, welches nicht direkt an einem Zustandssymbol anliegt, muss in der nächsten Zeile unverändert bleiben, auf Grund der Arbeitsweise von Turingmaschinen. Jedes dieser Bandsymbole ist das obere, mittige Element eines legalen Fensters. Die $\#$ Symbole können von keinem Konfigurationsübergang verändert werden, da sie nicht in N_P vorkommen. Von Zeile zu Zeile ändert sich dementsprechend nur ein einziger Teil, der durch ein einzelnes Fenster beschrieben werden kann. Nämlich das Fenster, in dessen obiger Mitte das Zustandssymbol steht. Ist dieses Fenster legal, wissen wir, dass die untere Zeile des Fensters aus der oberen folgen kann, und somit folgt die komplette untere Zeile aus der oberen. Da wir feststellen können, ob ein Fenster legal ist, können wir auch feststellen ob das Fenster mit dem Zustandssymbol in seiner obigen Mitte legal ist.
Unsere letzte Teilformel lässt sich ausdrücken als:

$$\varphi_{move} = \bigwedge_{1 \leq i < n^k, 1 < j < n^k} \left(\bigvee_{s_1,..,s_6 \text{sind legal}} \begin{matrix} x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge \\ x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6} \end{matrix} \right)$$

Wir bezeichnen Symbole $s_1, ..s_6$ als legal, wenn sie zu einem legalen Fenster gehören. Die Formel verlangt also, dass für jedes Bandsymbol jeder Zeile ($\#$ sind keine Bandsymbole, aber werden implizit mit verifiziert) ein legales Fenster existiert, sodass das Symbol in dessen obiger Mitte auftaucht. Kurzum, alle Fenster sind legal, wenn die Formel erfüllt ist.

Zum Nachdenken: *Wie sehen legale Fenster in der letzten Zeile aus, und benötigen wir diese für den Beweis? Warum haben wir die Fenstergröße auf* 2×3 *gewählt, und nicht z.B.* 2×2

oder 3×5?
Nachdem wir nun alle Teilformeln von φ beschrieben und uns von deren Korrektheit überzeugt haben, müssen wir noch zeigen, dass φ in polynomieller Zeit erzeugbar ist. Das ist der eigentliche Knackpunkt, denn unsere Reduktion soll ja nicht nur korrekt sein, sondern auch effizient genug arbeiten, dass wir nicht die Komplexitätsklasse unserer Probleme verlassen.
Um zu zeigen, dass wir φ in polynomiell vielen Schritten aufbauen können, betrachten wir φ's Länge in Relation zur Eingabe. Zunächst überlegen wir, wie viele Variablen in φ auftauchen. Sei $d = |C|$, dann d Variablen pro Zelle, macht bei bei Eingabelänge $n = |w|$ insgesamt $\mathcal{O}(n^{2k})$.
φ_{cell} enthält eine Konstante Größe pro Zelle, wir bleiben also in Größenordnung $\mathcal{O}(n^{2k})$.
φ_{start} ist $\mathcal{O}(n^k)$ lang, da es die erste Zeile des Tableaus kodiert.
phi_{move} und phi_{accept} beinhalten wie φ_{cell} jeweils Teilformeln konstanter Länge für jede Zelle des Tableaus, liegen also ebenfalls in $\mathcal{O}(n^{2k})$. Wenn wir alle vier Teilformeln aufsummieren, ändert sich daran nichts, also besitzt φ insgesamt eine Länge von $\mathcal{O}(n^{2k})$. Jedes Symbol der Formel kann schnell generiert werden.
Wir haben somit bewiesen, dass wir jede Sprache in $\mathcal{NP}$ mit SAT Formeln in polynomieller Zeit simulieren können, indem wir eine allgemeine Konstruktionsvorschrift für die entsprechende Reduktionsfunktion angegeben haben.
SAT ist $\mathcal{NP}$-vollständig. □

Nun, da wir für ein Problem ohne Hilfe von Reduktionen bewiesen haben, dass es $\mathcal{NP}$-vollständigist, können zukünftige Beweise auf dieser Erkenntnis aufbauen mit weniger Aufwand geführt werden. Als nächstes wollen wir ein Problem vorstellen, welches auf Grund seiner einfachen Struktur gerne für Reduktionen verwendet wird.

Definition 30.

$$3SAT = \{k_1, .., k_m \mid k_i \textit{ Klausel über } x_1, ..x_n \textit{ mit } |k_i| = 3 \wedge (\exists a_1, .., a_n \in \{0,1\}. \forall j \leq m. a_1, .., a_n \textit{ erfüllt } k_j)\}$$

Alternativ:

$$3SAT = \{\varphi \mid \varphi \textit{ ist eine erfüllbare Formel in KNF, die ausschließlich aus Klauseln mit 3 Literalen besteht}\}$$

Theorem 31 *$3SAT$ ist $\mathcal{NP}$-vollständig.*

Um die $\mathcal{NP}$-Vollständigständigkeit von $3SAT$ zu zeigen, könnten wir den Beweis des $COOK$-$LEVIN$-Theorems modifizieren, sodass φ nur aus Klauseln mit je 3 Variablen besteht. Da uns jedoch im Grunde nur die Arbeitsweise der Transformation von φ auf $3SAT\varphi$ interessiert, führen wir den Beweis über polynomielle Reduktion.

Beweis: $3SAT \in \mathcal{NP}$ - die NTM sucht sich nichtdeterministisch alle Variablenbelegungen aus, verifiziert sie in linearer Zeit und ist dementsprechend polynomiell und korrekt.
Wir zeigen, wie man eine beliebige SAT Formel in eine $3SAT$ Formel transformiert, und dass diese Transformation polynomiell ist.
Sei $x \in SAT$, dann ist x von der Form $x = k_1 \wedge .. \wedge k_n$, wobei jedes k_i eine Disjunktion von Literalen ist. Sei *f unsere Reduktionsfunktion*. Wir müssen zeigen, dass $\forall x \in SAT.\ f(x) \in 3SAT$. Wir beschreiben im Folgenden die Transformationsregeln:

Enthält x eine Klausel $k_i = (a)$ oder $k_i = (a \vee b)$ für Literale a, b, dann produzieren wir daraus eine 3-stellige Klausel, indem wir eines der Literale mehrmals in die Klausel schreiben. Z.B. $k_{i_f} = (a \vee a \vee a)$ oder $(a \vee b \vee b)$.
Enthält x eine Klausel mit Länge 3, lassen wir diese wie sie ist.
Enthält x eine Klausel mit mehr als 3 Literalen, zerlegen wir diese folgendermaßen: $k_i = (l_1 \vee .. \vee l_n)$ zu $(l_1 \vee l_2 \vee z_1) \wedge (\overline{z_1} \vee l_3 \vee z_2) \wedge .. \wedge (\overline{z_m} \vee l_{n-1} \vee l_n)$
Z.B. $k_i = (a \vee b \vee c \vee d \vee e \vee g) \rightarrow (a \vee b \vee z_1) \wedge (\overline{z_1} \vee c \vee z_2) \wedge (\overline{z_2} \vee d \vee z_3) \wedge (\overline{z_3} \vee e \vee g)$
Die abwechselnd positiven und negativen z_i verhindern, dass wir eine Formel nur durch eine Belegung aller z_i erfüllen könne (weswegen wir auch verschiedene z_i für je zwei verschiedene Klauseln benötigen). Diese Aufteilung ändert nichts an der Erfüllbarkeit der Formel, sie wird lediglich neu strukturiert. Dies ist leicht zu zeigen: war die ursprüngliche Klausel erfüllbar, so setzen wir das entsprechende Literal l, welches sie erfüllbar machte, auf 1. Dann können wir das z_i, welches in der gleichen Klausel auftritt wie l auf 0 setzen und erfüllen somit die darauf folgende Klausel mit $\overline{z_i}$. In dieser Formel taucht eventuell noch ein z_j auf, welches wir wiederum auf 0 setzen, da die Klausel bereits erfüllt ist, womit wir die nächste Klausel erfüllen. Die restlichen Klauseln können auf die gleiche Weise durch die verbleibenden z's erfüllt werden.
Bsp. Wir nutzen die gleiche Klausel wie eben. Sei $a = 1$, dann
$(1 \vee b \vee 0) \wedge (\overline{0} \vee c \vee 0) \wedge (\overline{0} \vee d \vee 0) \wedge (\overline{0} \vee e \vee g) = 1 \wedge 1 \wedge 1 \wedge 1$.
War die Klausel nicht erfüllbar, so können wir sie weiterhin nicht durch die originalen Literale erfüllen, und wir haben bereits festgestellt, dass die z_i für sich genommen auf Grund ihrer abwechselnden Vorzeichen keine Lösung erzeugen.
Nun, da wir die Reduktion haben, schätzen wir deren Laufzeit ab. Sie darf höchstens polynomiell sein, damit der Beweis funktioniert.
Jede Klausel der Eingabe x wird entweder belassen, um 1 oder 2 erweitert, oder durch $n - 2$ Klauseln der Länge 3 ersetzt, wobei n diesmal nicht die Länge von x, sondern die Anzahl der Literale der Klausel ist. In jedem Fall verlängert sich unsere Ergebnisformel maximal linear, und jede unserer Regeln ist in linearer Zeit anwendbar. Somit ist die Reduktion polynomiell.
$3SAT$ ist $\mathcal{NP}$-vollständig. □

Als nächstes beweisen wir die $\mathcal{NP}$-Vollständigkeit des Problems, das den Kern von KP, $PART$, BPP (siehe Vorlesungsfolien 6.2) und weiteren bildet: $SUBSET\text{-}SUM$.

Definition 32.

$$SUBSET\text{-}SUM = \{\langle S, t\rangle \mid S = \{x_1, .., x_k\} \wedge \exists \{y_1, .., y_p\} \subseteq S.\, \Sigma y_i = t\}$$

In Worten: Wir haben eine Menge an Zahlen S und möchten wissen, ob eine Teilmenge von S t ergibt, wenn man ihre Elemente aufsummiert. Falls eine solche Teilmenge existiert, gilt $\langle S, t\rangle \in SUBSET\text{-}SUM$.

Theorem 33 *$SUBSET\text{-}SUM$ ist $\mathcal{NP}$-vollständig.*

Beweis: Wir führen den Beweis über Reduktion von $3SAT$ auf $SUBSET\text{-}SUM$. Zunächst zeigen wir jedoch, dass $SUBSET\text{-}SUM \in NP$.
Sei N *eine NTM*. N akzeptiert $SUBSET\text{-}SUM$ folgendermaßen:
$N =$ " auf Eingabe $\langle S, t\rangle$ tue:

1. Prüfe, ob Eingabe in korrekter Form ist. Falls nicht, REJECT.
2. Wähle nichtdeterministisch alle Teilmenge von S aus.
3. Summiere die Elemente der ausgewählten Teilmenge auf.
4. Ist die Summe $= t$, ACCEPT. Sonst, REJECT. "

Im ersten Schritt muss N neben der Syntax der Eingabe prüfen, ob S eine Menge von Zahlen und t eine Zahl ist. Im zweiten Schritt wird jede Teilsumme von S nichtdeterministisch betrachtet, deren Elemente aufaddiert und schließlich mit t verglichen. Jeder dieser Schritte ist in linearer Zeit machbar. Somit ist N korrekt und polynomiell. Wir folgern, $SUBSET\text{-}SUM \in NP$.
Als nächstes formulieren wir eine Reduktion von $3SAT$ nach $SUBSET\text{-}SUM$. Das bedeutet, wir müssen eine in polynomieller Zeit berechenbare Funktion $f : 3SAT \rightarrow SUBSET\text{-}SUM$ beschreiben, sodass gilt: $x \in 3SAT \Leftrightarrow f(x) \in SUBSET\text{-}SUM$. Die Richtung ist dabei entscheident. Wir reduzieren das Problem, zu entscheiden ob ein Element x in $3SAT$ enthalten ist, auf das Problem, ob das entsprechende Element $f(x)$ in $SUBSET\text{-}SUM$ enthalten ist. Das bedeutet, dass $3SAT$ nicht schwerer sein kann als $SUBSET\text{-}SUM$: $3SAT \leq SUBSET\text{-}SUM$. Da $3SAT$ bereits zur schwersten Problemklasse in $\mathcal{NP}$ gehört, wissen wir dadurch, dass auch $SUBSET\text{-}SUM$ $\mathcal{NP}$-vollständigsein muss.
Wir stellen die Reduktion zunächst vor, argumentieren, dass sie korrekt ist, und analysieren schließlich ihre Laufzeit.
Sei *φ eine boolesche Formel in KNF* bestehend aus *Variablen $x_1, .., x_v$* und *Klauseln $c_1, .., c_k$*. Wir konstruieren aus φ eine Instanz $\langle S, t\rangle$ des $SUBSET\text{-}SUM$ Problems. Für jede Variable x_i aus φ hat S zwei Elemente $\top_i$ und $\bot_i$, wobei der Zahlwert von $\top_i$ zu unserer Summe gehört, wenn x_i zu einer entsprechenden erfüllenden Belegung von φ gehört, und der Zahlwert von $\bot_i$, wenn nicht. Die Zahl, mit der wir $\top_i$ bzw. $\bot_i$ darstellen, setzt sich aus zwei Teilen zusammen. Der vordere Teil ist eine 1 gefolgt von $v - i$ Nullen. Der zweite Teil setzt sich aus sovielen Ziffern zusammen, wie es Klauseln gibt, wobei eine Ziffer auf 1 gesetzt wird, wenn die entsprechende Klausel $\top_i$ bzw. $\bot_i$ enthält. Andernfalls wird die Ziffer auf 0 gesetzt.
Desweiteren enthält S zwei Elemente $fillc_j$ und $fillC_j$ für jede Klausel c_j aus φ. Die Zahlwerte sind eine 1 gefolgt von $k - j$ Nullen. Diese Elemente helfen uns später auf unsere vordefinierte Summe zu kommen (siehe Abb. 5).
Nun, da wir S beschrieben haben, kommen wir zur gewünschten Summe t. Wir nennen sie in den Graphiken $\sum_\varphi$. Wir verlangen, dass $\sum_\varphi$ von folgender Form ist: Die ersten v Ziffern müssen Einsen sein, die restlichen k Ziffern müssen Dreien sein. Die Summe darf weder kürzer noch länger, weder niedriger noch höher sein. Die ersten v Ziffern erzwingen, dass jede Variable in unserer Formel φ entweder mit TRUE oder FALSE belegt ist. Wäre beides belegt, würden wir eine 2 statt 1 in der Summe stehen haben, wäre keines von beiden belegt, würde eine 0 stehen. Beide Fälle können nicht auftreten, wenn die Variablenbelegung eine erfüllende ist. Die letzten k Ziffern der Summe legen fest, dass in jeder Klausel maximal 3 wahre Literale stehen dürfen. Wenn es weniger sind, können wir unsere Summe mit Hilfe von $fillc_j$ oder $fillC_j$ an der entsprechenden Stelle auf 3 erhöhen. Wenn es mehr sind, steht an der entsprechenden Stelle in der Summe eine 4. Da unsere Klauseln jedoch maximal 3 Literale haben (wir reduzieren schließlich von $3SAT$), kann dieser Fall nicht eintreten. Andernfalls war unsere Eingabeformel kein Element von $3SAT$.

	1	2	3	4	··	v	c_1	c_2	··	c_k
$\top_1$	1	0	0	0	·	0	?	?	··	?
$\bot_1$	1	0	0	0	·	0	·			·
$\top_2$		1	0	0		0	·			·
$\bot_2$		1	0	0		0				
$\top_3$			1	0		0				
$\bot_3$			1	0		0				
·				·		0				
·				·		0				
$\top_v$						1				
$\bot_v$						1				
$fillc_1$							1	0	0	0
$fillC_1$							1	0	0	0
$fillc_2$								1	0	0
$fillC_2$								1	0	0
·									1	0
·									1	0
$fillc_k$										1
$fillC_k$										1
$\sum_\phi$	1	1	1	1	··	1	3	3	··	3

Abbildung 5. Allgemeiner Aufbau der Elemente von S

Betrachten wir folgendes Beispiel (Abb. 6): $\varphi = (x_1 \vee x_2 \vee \neg x_5) \wedge (x_4 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_4 \vee x_2 \vee \neg x_5) \wedge (x_5 \vee x_1 \vee x_3)$. Unsere fünf Variablen sind $x_1, .., x_5$, wir haben vier Klauseln (in Reihenfolge in φ) $c_1, .., c_4$. Nach Transformation in die vorgestellte $SUBSET$-SUM Form, können wir folgende rot markierte Elemente aus S wählen, um unsere gewünschte Summe zu erhalten: Wir schließen daraus, dass φ erfüllbar ist, und zwar mit der Belegung $x_1, \neg x_2, x_3, \neg x_4, \neg x_5$, entsprechend unserer ausgewählten Elemente $\top_1, \bot_2, \top_3, \bot_4, \bot_5$.
Wir haben bereits motiviert, weshalb diese Konstruktion funktioniert, und führen diese Überlegungen nun präziser aus.
Angenommen, φ ist erfüllbar, d.h. $\varphi \in 3SAT$. Dann wählen wir für jede positive Belegung von x_i unser Element $\top_i \in S$ aus, für jede negative Belegung $\bot_i \in S$. Da wir pro Variable nur entweder das eine oder andere Element auswählen, muss an der i-ten Stelle unserer Summe wie gefordert eine 1 stehen. Dies muss der Fall für alle Variablen, also für die ersten v Ziffern der Summe sein. Da diese Variablenbelegung φ nach Voraussetzung erfüllt, muss jede Klausel c_j erfüllt sein. Nach Konstruktion unserer Elemente von S bedeutet dies, dass jede der letzten k Ziffern unserer Summe mindestens eine 1, maximal eine 3 ist. Falls es weniger als 3 sind, fügen wir unserer Summe das entsprechende $fillc_j$ und/oder $fillC_j$ hinzu, um auf 3 zu kommen. Unsere Summe ist nun in der geforderten Form, somit ist $\langle S, t\rangle \in SUBSET$-$SUM$.
Nun zur Gegenrichtung.
Angenommen eine Teilmenge von S summiert zu t auf. Wir konstruieren die erfüllende Belegung für unsere Formel folgendermaßen: Für jedes $\top_i$ belegen wir x_i mit TRUE, für

						c_1	c_2	c_3	c_4
$\top_1$	1	0	0	0	0	1	0	0	1
$\bot_1$	1	0	0	0	0	0	0	0	0
$\top_2$		1	0	0	0	1	0	1	0
$\bot_2$		1	0	0	0	0	1	0	0
$\top_3$			1	0	0	0	0	0	1
$\bot_3$			1	0	0	0	1	0	0
$\top_4$				1	0	0	1	0	0
$\bot_4$				1	0	0	0	1	0
$\top_5$					1	0	0	0	1
$\bot_5$					1	1	0	1	0
$fillc_1$						1	0	0	0
$fillC_1$						1	0	0	0
$fillc_2$							1	0	0
$fillC_2$							1	0	0
$fillc_3$								1	0
$fillC_3$								1	0
$fillc_4$									1
$fillC_4$									1
$\sum_\phi$	1	1	1	1	1	3	3	3	3

Abbildung 6. In rot umrandet die ausgewählten Elemente unserer Summe.

jedes $\bot_i$ belegen wir x_1 mit FALSE. Da unsere Auswahl korrekt aufsummiert, wissen wir, dass jede Variable mit genau einem Wert belegt wurde. Andernfalls hätte in unserer Summe eine 0 oder 2 gestanden, womit die aufaddierte Teilmenge nicht gleich t gewesen wäre. Dies widerspräche unserer Voraussetzung. Da die letzten k Ziffern unserer Summe 3 sind, und wir mit $fillc_j$ und $fillC_j$ höchstens 2 aufaddieren können, muss mindestens eine 1 von jeweils einem der ausgewählten Variablenelemente sein. Das bedeutet, jede unserer Klauseln ist durch mindestens eine Variablenbelegung erfüllt. Somit ist $x_1, .., x_v$ nach unserer Belegung eine erfüllende Belegung für φ.
Da unsere Konstruktion auf der Addition großer Zahlen basiert, müssen wir noch darauf achten, dass kein Übertrag unsere Konstruktion ausversehen verfälschen kann. Da in jeder Ziffernspalte maximal fünf Einsen vorkommen, kann es nie zu einem Übertrag kommen.
Unsere Reduktionsfunktion ist korrekt. Zuletzt schätzen wir ihre Laufzeit ab, um sicherzustellen, dass sie in polynomieller Zeit arbeitet.
Unsere Reduktion lässt sich als einfache Matrix darstellen, deren einzelne Elemente leicht berechnet werden können. Die Matrix hat lediglich $(v + k) * (2 * v + 2 * k + 1)$ Zellen, die berechnet werden müssen. Wir bestimmen das Quadrat der Eingabelänge als obere Schranke für die Laufzeit unserer Reduktion, womit sie polynomiell ist.
$SUBSET\text{-}SUM$ ist $\mathcal{NP}$-vollständig.

□

Im Beweis für $SUBSET\text{-}SUM$ haben wir gesehen, dass wir mit der Reduktion von $3SAT$ die Eigenschaft ausnutzen konnten, dass jede Klausel exakt 3 Literale enthält. Wir wussten

von vorneherein ganz genau, wie unsere Summe diesem gerecht werden kann. In vielen Fällen erleichtert diese Eigenschaft von $3SAT$ Reduktionsbeweise. Betrachten wir im Folgenden einen $\mathcal{NP}$-Vollständigkeitsbeweis, bei dem dies keine Rolle spielt.

Theorem 34 *$HAMPATH$ ist $\mathcal{NP}$-vollständig.*

Beweis: Wir führen den Beweis über polynomielle Reduktion von SAT. Wir haben bereits in Beispiel 22 gezeigt, dass $HAMPATH \in \mathcal{NP}$.
In vielen Quellen wird der $\mathcal{NP}$-Vollständigkeitsbeweis von $HAMPATH$ über Reduktion von $3SAT$ geführt. Reduktionen von $3SAT$ machen in vielen $\mathcal{NP}$ Beweisen Sinn, wie wir bereits erläutert haben. In diesem Beweis interessiert uns jedoch nur, dass die Eingabeformeln in konjunktiver Normalform stehen, was bereits von SAT erfüllt wird. Wir könnten den Beweis $3SAT \leq_P HAMPATH$ also im Prinzip analog zu diesem führen, indem wir jedes SAT durch $3SAT$ ersetzen.
Wir zeigen, wie wir aus einer Formel $\varphi \in SAT$ einen Graphen G konstruieren, sodass G einen hamilton'schen Pfad von einem s nach t besitzt.
Sei *φ eine Formel in KNF* mit *Klauseln* $c_1, .., c_k$ und *Variablen* $x_1, .., x_v$. Wir repräsentieren jede Klausel c_j durch einen einzelnen Knoten in unserem Graphen. Wir beschreiben deren Kanten später. Jede Variable x_i wird durch die in Abbildung 7 dargestellte Struktur repräsentiert.

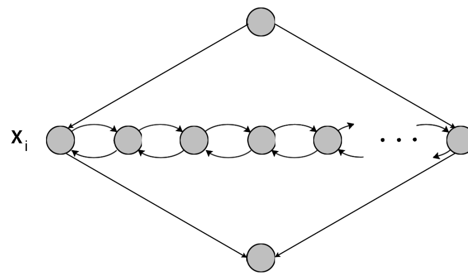


Abbildung 7. Jede Variable x_i wird durch diese Struktur dargestellt.

Der hamilton'sche Pfad geht von oben nach unten, wobei er links-rechts verläuft, wenn x_i TRUE ist, und rechts-links, wenn x_i FALSE ist (siehe Abb.8). Ohne zu spezifizieren, wie viele Knoten pro Variable benötigt und wie die Klauselknoten verbunden werden, sieht unser Gesamtgraph G wie in 9 dargestellt aus. Da der gesuchte Pfad von s nach t verlaufen muss, muss er von oben nach unten durch die einzelnen Variablenabschnitte gehen. Für jede Variable wird dabei festgelegt, ob diese positiv oder negativ vorkommt, indem der Pfad entweder links-rechts oder rechts-links verläuft. In jedem Fall muss der Pfad alle Knoten jedes Variablenkonstrukts durchlaufen und im untersten Knoten t enden. Nun erzwingen wir, dass die durchlaufende Variablenbelegung einer erfüllenden Belegung entspricht, indem wir entsprechende Kanten zu den Klauselknoten hinzufügen.
Die Mittelreihe jeder Variablenkonstruktion enthält $3k+3$ Knoten. Zwei davon sind die beiden äußersten, also bleiben $3k+1$ innere Knoten. Wir assoziieren diese inneren Knoten mit den Klauselknoten wie dargestellt in Abb. 10. Wir verbinden diese Knoten mit den Klausel-

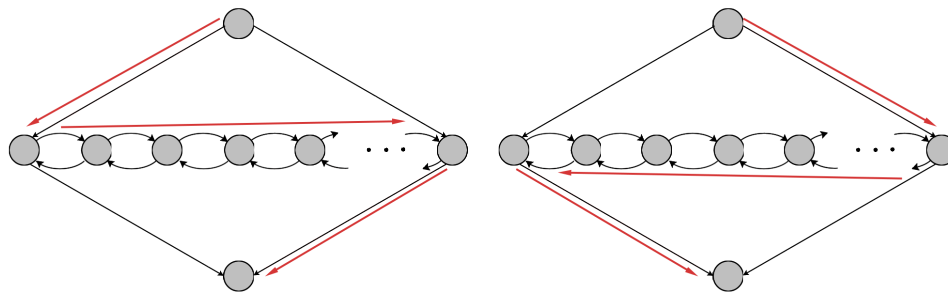


Abbildung 8. Links: links-rechts Orientierung des hamilton'schen Pfades. Rechts: rechts-links Orientierung des hamilton'schen Pfades.

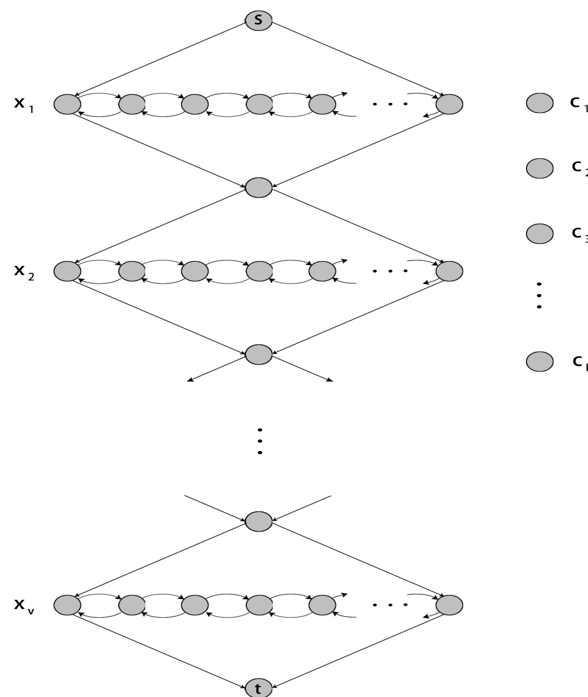


Abbildung 9.

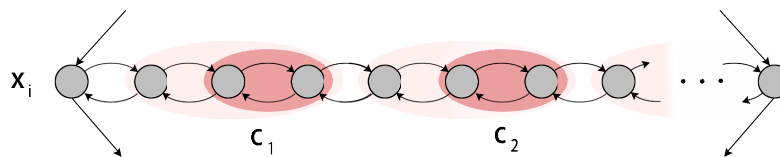


Abbildung 10.

knoten entsprechend x_i's Vorkommen in Klausel c_j. Wir kommen von links nach rechts in den Klauselknoten und zurück, falls x_i in c_j auftaucht, und von rechts nach links, wenn $\neg x_i$ in c_j vorkommt. Siehe Abb. 11.

Hiermit ist unsere Konstruktionsvorschrift für G vollständig. Wir zeigen nun, dass sie kor-

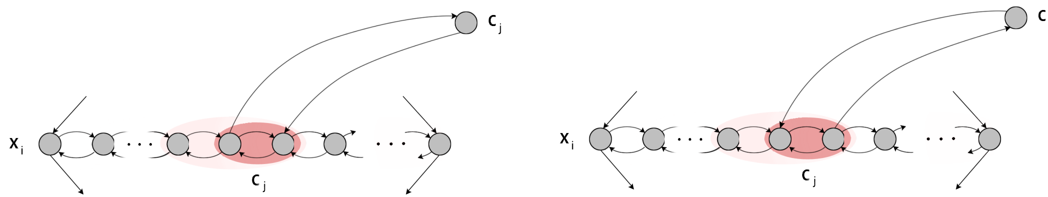


Abbildung 11. Links: G enthält diese Kanten, wenn x_i in c_j vorkommt. Rechts: G enthält diese Kanten, wenn $\neg x_i$ in c_j vorkommt

rekt ist.

Sei $\varphi \in SAT$, dann finden wir in G den hamilton'schen Pfad folgendermaßen. Für jede positive Variable x_i, die zu φ's erfüllender Belegung gehört, führt der Pfad links-rechts durch das i-te Variablenkonstrukt. Für jede negative Variable stattdessen von rechts nach links. Somit haben wir bereits alle Knoten besucht, außer solche, die einer Klausel zugeordnet werden. Da φ erfüllbar ist, ist jede Klausel erfüllt. Das heißt, für jeden Klauselknoten gibt es mindestens ein Variablenkonstrukt, von welchem wir den Klauselknoten aus erreichen und wieder besuchen können. Auf Grund der Konstruktion in 9 übersehen wir dabei keinen Knoten unserer Variablenkonstrukte. Somit haben wir einen Pfad beschrieben, der alle Knoten in G genau einmal besucht.

Für die Rückrichtung nehmen wir an, dass G einen hamilton'schen Pfad von s nach t enthält. Wir zeigen, wie wir eine erfüllende Variablenbelegung daraus ablesen. Wenn der Pfad von oben nach unten durch alle Variablenkonstrukte und Klauselknoten verläuft, ist unsere Variablenbelegung für x_i daran ablesbar, ob der Pfad im entsprechenden Konstrukt von links nach rechts oder von rechts nach links verläuft. Offensichtlich ist diese Belegung erfüllend, da jede Klausel von mindestens einer Belegung erfüllt ist. Zu zeigen ist, dass kein anderer hamilton'scher Pfad in G existieren kann.

Der Pfad muss von s nach t verlaufen, das bedeutet, die einzige Möglichkeit, wie ein Pfad aussehen kann, wenn er nicht wie beschrieben verläuft, ist, dass er die Variablenkonstrukte nicht alle in der Reihenfolge von oben nach unten durchläuft. Dies kann nur der Fall sein, wenn der Pfad über einen Klauselknoten ein anderes Variablenkonstrukt besucht als jenes, von dem er kam. Siehe Abb. 12.

In diesem Fall kann der Pfad jedoch nicht alle Knoten des Graphen besucht haben, da es keine Möglichkeit gibt z_2 zu erreichen und von dort aus nach t zu laufen. Denn die einzigen Möglichkeiten z_2 zu erreichen sind folgende: Entweder gehen nur je ein Pfad von z_1, c, z_3 auf z_2, oder nur von z_1, z_3. z_1, c sind jedoch bereits besucht worden und dürfen nicht nocheinmal benutzt werden. Das heißt, die einzige Möglichkeit z_2 legal zu erreichen ist über z_3, doch von z_2 führt nur je ein Pfad auf z_1, z_3 und eventuell c. Sobald z_2 erreicht wurde, sind alle drei Nachbarn jedoch bereits besucht worden, sodass der Pfad nicht mehr

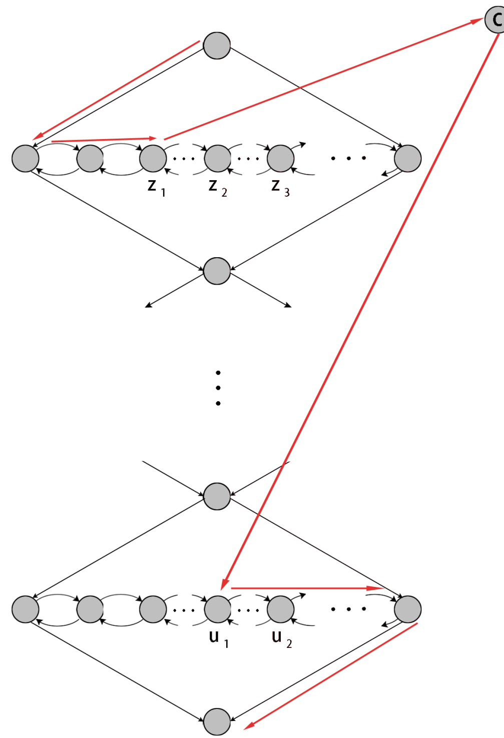


Abbildung 12.

fortgesetzt werden kann. Der Pfad muss aber in t enden, was einen Widerspruch erzeugt. Wir folgern, dass es in G keinen anderen hamilton'schen Pfad als den zuvor beschriebenen geben kann.
Unsere Reduktion ist dementsprechend korrekt, zuletzt analysieren wir ihre ungefähre Laufzeit, um sicherzustellen, dass sie auch polynomiell ist. Wir haben $v * (3 + 3 * k + 1) + 1 + k$ Knoten im Graphen, und etwa gleichviele Kanten. Jeder Knoten und jede Kante ist eindeutig definiert. Den Graphen zu bauen ist in in $\mathcal{O}(n)$ machbar, wobei n die Länge der Eingabeformel ist. Somit ist unsere Reduktion polynomiell.
$HAMPATH$ ist $\mathcal{NP}$-vollständig.

□

Wir wollen das Kapitel um die Klassen $\mathcal{P}$ und $\mathcal{NP}$ hiermit schließen. Wir haben herausgefunden, dass $\mathcal{P}$ die Klasse aller Probleme ist, die in polynomieller Zeit auf einer deterministischen Turingmaschine, und damit auch auf modernen Computern, relativ leicht lösbar sind. Auf der anderen Seite haben wir $\mathcal{NP}$ vorgestellt, die Klasse der Probleme, die auf nichtdeterministischen Modellen in polynomieller Zeit lösbar sind. Bislang kennen wir keine effizienten Algorithmen (d.h. polynomielle), um $\mathcal{NP}$ zu lösen - bemerkenswerterweise beinhaltet $\mathcal{NP}$ jedoch Probleme durch deren Lösung wir alle ihre Probleme lösen können. Wir nennen diese $\mathcal{NP}$-vollständig, und würde man einen Algorithmus für auch nur eines dieser Probleme finden, könnten wir die $\mathcal{P} = \mathcal{NP}$ Frage positiv beantworten. Beweist man widerum, dass irgendein $\mathcal{NP}$-vollständiges Problem schlichtweg nicht durch einen deterministischen, polynomiellen Algorithmus gelöst werden kann, so hat man dies für alle anderen $\mathcal{NP}$-vollständigen Probleme

ebenfalls bewiesen. Wir wüssten in diesem Fall, dass $\mathcal{P} \neq \mathcal{NP}$ gilt.

Literatur

1. Michael Sipser. *Introduction to the Theory of Computation*. 12
2. Jeffrey D. Ullman John E. Hopcroft, Rajeev Motwani. *Einführung in die Automatentheorie, Formale Sprachen und Berechenbarkeit*. 7