Computational Complexity – Komplexitätstheorie

Domenic Cassisi
Fakultät Informatik
Hochschule Furtwangen
Furtwangen im Schwarzwald, Deutschland
domenic.cassisi@hs-furtwangen.de

Thomas Hass
Fakultät Informatik
Hochschule Furtwangen
Furtwangen im Schwarzwald, Deutschland
thomas.hass@hs-furtwangen.de

***Zusammenfassung*—Diese Arbeit behandelt grundlegende Aspekte der Komplexitätstheorie und veranschaulicht vorgestellte Konzepte anhand konkreter Beispiele. Die O-Notation dient als Grundlage zur Bildung von Komplexitätsklassen, insbesondere für die Zeitkomplexitätsanalyse. Bei der Beschreibung der Zeitkomplexität eines Algorithmus lassen sich sowohl Worst-, Best- und Average-Case-Analysen durchführen, wobei insbesondere letztere aufwändiger sind, jedoch in der Praxis eine wichtige Rolle einnehmen. Eine breitere Klassifizierung der Probleme und ihren Algorithmen ermöglichen die Komplexitätsklassen P, NP, NP-vollständig und NP-schwer. Das bekannte P-NP-Problem ist bis heute ungelöst, wobei mittlerweile davon ausgegangen wird, dass die beiden Komplexitätsklassen verschieden sind. Um effizient Optimallösungen für schwere Probleme anzunähern, können Heuristiken oder Approximationsalgorithmen verwendet werden. Heuristiken führen häufig zu guten Ergebnissen, garantieren jedoch keine Güte. Alternativ eignen sich Approximationsalgoritmen, wenn eine bestimmte Gütegarantie für das berechnete Ergebnis erforderlich ist. Als Ausblick betrachtet die Arbeit den Einfluss programmierbarer Quantencomputer auf die Komplexitätstheorie. Dabei ist festzustellen, dass Quantencomputer prinzipiell die Lösung vieler schwerer Probleme beschleunigen könnten, jedoch bis heute unklar ist, ob sich dadurch NP-vollständige Probleme in polynomieller Laufzeit lösen lassen.**

***Index Terms*—Komplexitätstheorie, Laufzeitkomplexität, Average-Case, P-NP-Problem, Approximationsalgorithmen, Quantencomputing**

I. Einleitung

Die Komplexitätstheorie ist neben der Berechenbarkeitstheorie und der Theorie der Formalen Sprachen eine der drei Säulen der Theoretischen Informatik und wird als Fortsetzung der Berechenbarkeitstheorie angesehen [1].

In den sechziger Jahren breitete sich die Nutzung von Rechenmaschinen aus, sodass nicht mehr nur Forschungsinstitute Zugriff zu Rechnern hatten. Dies stellte viele Menschen vor die Erkenntnis, dass die Existenz eines Algorithmus, der ein bestimmtes Problem löst, nicht zwingend bedeutet, dass der Algorithmus auch erfolgreich auf dem Rechner ausgeführt wird. Oft liefen Programme mehrere Tage und letztlich stürzte der Rechner ab, bevor eine Lösung berechnet werden konnte. Dadurch entstand Unsicherheit, ob der Algorithmus fehlerhaft war oder es sich um eine inhärente Eigenschaft des Problems handelt, die keine effiziente algorithmische Lösung zulässt [1].

Alan Cobham zeigte 1964 die Lösbarkeit vieler wichtiger Probleme in Polynomialzeit. 1971 führte Stephan Cook den Begriff der NP-Vollständigkeit ein, wodurch die P-NP-Problematik aufkam, die bis heute nicht geklärt ist und in dieser Arbeit aufgegriffen wird [2]. Die Komplexitätsklassen haben sowohl für Theoretiker als auch in der Praxis für Entwickler:innen eine hohe Bedeutung, da sie trotz Weiterentwicklung der Rechenleistung bestehen bleiben und sie das entscheidende Maß darstellen, um die Komplexität von Problemen zu beschreiben [1].

A. Ziele der Komplexitätstheorie

Hromkovic definiert in seinem Werk [1] die Hauptziele der Komplexitätstheorie wie folgt:

1) Die Bestimmung der Berechnungskomplexitäten konkreter Probleme.
2) Die Spezifikation der Klasse effizient lösbarer Probleme sowie die Entwicklung von Methoden zur Klassifkation der algorithmisch lösbaren Probleme in „praktisch lösbare“und „praktisch unlösbare“.
3) Vergleiche der Effizienz von deterministischen, nichtdeterministischen und zufallsgesteuerten Algorithmen.

Im Laufe dieser Arbeit wird insbesondere auf die ersten beiden Ziele eingegangen. Das dritte Ziel ist ein sehr fortgeschrittenes und umfangreiches Themengebiet der Komplexitätstheorie und wird in dieser Arbeit aus Gründen des Umfangs nicht näher behandelt.

B. Ziel der Arbeit

Ziel dieser Arbeit ist zu zeigen, wie sich die im letzten Abschnitt ausgewählten Hauptziele der Komplexitätstheorie durch konkrete Methoden erreichen lassen. Zudem sollen die in der Arbeit vorgestellten Konzepte anhand konkreter Beispiele verdeutlicht werden.

C. Struktur der Arbeit

Die Arbeit ist wie folgt aufgebaut: In Abschnitt II wird die O-Notation als Methodik zur Beschreibung der Komplexität von Algorithmen vorgestellt und darauf aufbauend das Vorgehen einer Zeitkomplexitätsanalyse präsentiert. Abschnitt III behandelt die wichtigsten Komplexitätsklassen und beschreibt die Bedeutung des P-NP-Problems. Abschnitt IV erläutert die unterschiedlichen Lösungsoptionen für Probleme, die in NP liegen. Dabei werden die Approximationsalgorithmen vertieft behandelt. Abschließend gibt Abschnitt V einen Ausblick über die Auswirkungen des Quantencomputings auf die Komplexitätstheorie.

II. Laufzeitkomplexität

Die *Zeitkomplexität*, auch Laufzeitkomplexität genannt, beschreibt das zeitliche Verhalten eines Algorithmus, abhängig von der Größe der Eingabe [3]. Dabei wird die Anzahl der elementaren Operationen untersucht, die ein Algorithmus benötigt, um ein Problem abhängig von der Eingabegröße zu lösen. Die Grundlage für die Analyse und Betrachtung der Zeitkomplexität bildet ein formales Maschinenmodell, zumeist eine deterministische Turingmaschine.

Platzkomplexität beschreibt, wie viel zusätzlicher Speicherplatz ein Algorithmus in Abhängigkeit von der Eingabegröße benötigt. Dabei sei anzumerken, dass der benötigte Speicherplatzbedarf für die Eingabedaten nicht berücksichtigt wird. In den folgenden Abschnitten wird jedoch nur die Laufzeitkomplexität genauer betrachtet. Grund dafür ist, dass die Beschreibung des Speicherplatzbedarfs im Gegensatz zum Laufzeitverhalten mittlerweile stark an Bedeutung verloren hat; Speicherplatz wird günstiger, während CPUs pro Kern nicht mehr merklich schneller werden und damit eine wertvolle Ressource darstellen.

Die Angabe von absoluten Laufzeiten zur Beschreibung des Laufzeitverhaltens eines Algorithmus ist allerdings nur bedingt sinnvoll, da die gemessenen Werte unter anderem von der verwendeten CPU abhängen. Zudem ist die Aussagekraft über das Laufzeitverhalten nur sehr schwach, da die gesamte Menge an möglichen Eingaben nur selten berücksichtigt werden kann [3], [4]. In der Komplexitätstheorie hat sich daher ein abstraktes Maß zur Beschreibung der Komplexität eines Algorithmus etabliert, die O-Notation, welche im folgenden Abschnitt näher betrachtet wird.

A. O-Notation

Die O-Notation (auch Big-O-Notation, Landau-Notation oder O-Kalkül) ist ein Maß, um das asymptotische Verhalten eines Algorithmus in Abhängigkeit von der Eingabegröße zu charakterisieren. Dabei wird primär untersucht, wie sich ein Algorithmus im schlimmsten Fall verhält [3]. Die O-Notation eignet sich gleichermaßen zur Beschreibung der Zeitkomplexität als auch der Platzkomplexität. Die grundlegende Idee der O-Notation ist die Beschreibung der Komplexität eines Algorithmus durch eine Funktion. Der Funktion wird dabei die Größe des Problems als Eingabeparameter übergeben. Um die Komplexität von Algorithmen miteinander vergleichen zu können, werden die entsprechenden Funktionen der Algorithmen, wie in folgender Definition gezeigt, durch Ordnungen klassifiziert [3], [5].

Es seien $f, g : \mathbb{N}_0 \to \mathbb{R}_+$ zwei Funktionen. Dabei stellt $\mathbb{N}_0$ die Größe des Problems dar. Die Funktion f ist in der Ordnung von g, falls es eine Konstante $c > 0$ und eine natürliche Zahl $n_0 \in \mathbb{N}_0$ gibt, sodass gilt:

$$f(n) \leq c \cdot g(n) \text{ für alle } n \geq n_0$$

Die Menge aller Funktionen, die von der Ordnung g sind, wird durch $\mathcal{O}(g)$ wie folgt beschrieben:

$$\mathcal{O}(g) = \{\ f : \mathbb{N}_0 \to \mathbb{R}_+ \ \mid\ f \text{ ist in der Ordnung von } g\ \}$$

Ist eine Funktion f in der Ordnung von g, gilt also $f \in \mathcal{O}(g)$, dann wird das Wachstum von f durch g beschränkt [6]. Das bedeutet, dass alle Funktionswerte von f ab einer festen Stelle n_0 kleiner als die entsprechenden Funktionswerte von g sind, wobei von der linearen Skalierung durch die Konstante c abgesehen wird [5].

Exemplarisch wird die Funktion $f(n) = 4n^2 + 8n + 10$ nach oben abgeschätzt und gezeigt, dass diese in der Ordnung $\mathcal{O}(n^2)$ enthalten ist:

$$\begin{aligned} f(n) &= 4n^2 + 8n + 10 \\ &\leq 4n^2 + 8n^2 + 10n^2 \\ &\leq 10n^2 + 10n^2 + 10n^2 \\ &\leq 30n^2 \\ &= 30 \cdot g(n) \end{aligned}$$

Die Abschätzung nach oben zeigt, dass es eine Konstante $c = 30 > 0$ und ein $n_0 = 1$ gibt, sodass $f(n) \leq c \cdot g(n)$ für alle $n \geq n_0$ gilt. Damit ist gezeigt, dass $f(n) = 4n^2 + 8n + 10 \in \mathcal{O}(n^2)$ gilt.

Die am häufigsten auftretenden Ordnungsfunktionen im Zusammenhang mit der Zeitkomplexitätsanalyse sind in Tabelle I dargestellt [5].

Tabelle I
Wichtige Zeitkomplexitäten mit ihren Ordnungsfunktionen

Wachstum	**Ordnung**
konstant	$\mathcal{O}(1)$
logarithmisch	$\mathcal{O}(log(n))$
linear	$\mathcal{O}(n)$
log-linear	$\mathcal{O}(n \cdot log(n))$
polynomiell	$\mathcal{O}(n^k), k \geq 2$
exponentiell	$\mathcal{O}(d^n), d \geq 1$
fakultät	$\mathcal{O}(n!)$

In Anlehnung an Tobias Günther [7] sind in Tabelle II die Auswirkungen einiger Zeitkomplexitäten auf die entsprechenden Laufzeiten für unterschiedliche Problemgrößen dargestellt. Dabei wird die Annahme getroffen, dass die Durchführung eines Schrittes eine Mikrosekunde in Anspruch nimmt.

Tabelle II
Laufzeitenvergleich unterschiedlicher Zeitkomplexitäten

Annahme: 1 Schritt dauert 1 µs = $10^{-6}s$					
Ordnung	**10**	**20**	**30**	**40**	**50**
$\mathcal{O}(1)$	1 µs	1 µs	1 µs	1 µs	1 µs
$\mathcal{O}(n)$	10 µs	20 µs	30 µs	40 µs	50 µs
$\mathcal{O}(n^2)$	100 µs	400 µs	900 µs	1,6 ms	2,5 ms
$\mathcal{O}(n^3)$	1 ms	8 ms	27 ms	64 ms	125 ms
$\mathcal{O}(2^n)$	1 ms	1 s	18 min	13 T	36 J
$\mathcal{O}(3^n)$	59 ms	58 min	6,5 J	3855 Jh	10^8 Jh
$\mathcal{O}(n!)$	3.62s	771 Jh	> Alter des Universums		

Die Tabelle macht deutlich, dass insbesondere bei Zeitkomplexitäten mit exponentieller sowie fakultativer Laufzeit

bereits für kleine Problemgrößen extrem hohe Laufzeiten entstehen. Algorithmen, deren Laufzeit sich durch ein Polynom beschreiben lassen, haben zumeist eine für die Praxis akzeptable Laufzeit [5]. Jedoch sind für bestimmte Probleme bis heute nur Algorithmen bekannt, die sich mit exponentieller Komplexität auf einer deterministischen Turingmaschine lösen lassen. Diese besonders schwer lösbaren Probleme werden in Abschnitt III gesondert betrachtet.

B. Average-, Best- und Worst-Case-Laufzeitanalyse

Im vorherigen Abschnitt wurde die O-Notation genutzt, um obere Schranken für das Laufzeitverhalten von Algorithmen zu definieren. Damit wird gleichermaßen das schlechteste Laufzeitverhalten betrachtet. Diese pessimistische Betrachtung gibt Aussagen über die Worst-Case-Komplexität eines Algorithmus. Es gibt jedoch gute Gründe, auch andere Komplexitätsarten in Betracht zu ziehen. So muss die Laufzeitkomplexität nicht ausschließlich von der Problemgröße abhängen. Es ist möglich, dass für verschiedene Eingabemengen der gleichen Problemgröße ein unterschiedliches Laufzeitverhalten auftritt. Zudem ist es denkbar, dass der Worst-Case nur auf ‚wenige' Eingaben zutrifft, jedoch sich für alle anderen Eingaben ein ganz anderes Verhalten beobachten lässt [4]. Die damit verbundene hohe Praxisrelevanz ist die Motivation, in diesem Abschnitt weitere Komplexitätsarten zu betrachten.

Die Idee der verschiedenen Komplexitätsarten ist in der Literatur gut beschrieben. So beschreibt Katoen in [8] die drei Komplexitätsarten für einen gegebenen Algorithmus A wie folgt:

Worst-Case: Die Worst-Case-Laufzeit von A ist die von A maximal benötigte Anzahl elementarer Operationen auf einer beliebigen Eingabe der Länge n.

Best-Case: Die Best-Case-Laufzeit von A ist die von A minimal benötigte Anzahl elementarer Operationen auf einer beliebigen Eingabe der Länge n.

Average-Case: Die Average-Case-Laufzeit von A ist die von A durchschnittlich benötigte Anzahl elementarer Operationen auf einer beliebigen Eingabe der Länge n.

Die Analyse hängt also maßgeblich von der Wahl der elementaren Operationen ab. Der ‚Vergleich zweier Zahlen' beim Sortieren eines Arrays oder bei der linearen Suche stellt beispielsweise eine elementare Operation dar [8].

Um die Komplexitätsarten formaler besser greifen zu können, seien folgende Definitionen gegeben, wobei n die Problemgröße und I eine konkrete Eingabe darstellt:

$$\begin{aligned} D_n &= \text{ Menge aller Eingaben der Länge } n \\ t(I) &= \text{ für } I \text{ benötigte Anzahl elementarer Operationen} \\ Pr(I) &= \text{ Wahrscheinlichkeit, dass Eingabe } I \text{ auftritt} \end{aligned}$$

Der Wert für $t(I)$ ergibt sich durch Analyse des Algorithmus, etwa durch Betrachtung einer Implementierung, wie nachfolgend am Beispiel in Abschnitt II-C gezeigt. Für $Pr(I)$ wird sich häufig an Erfahrungswerten bedient, oder die Wahrscheinlichkeit abgeschätzt [8]. Die drei Komplexitätsarten lassen sich formal nun wie folgt definieren:

$$\begin{aligned} W(n) &= \max\{\, t(I) \mid I \in D_n \,\} \\ B(n) &= \min\{\, t(I) \mid I \in D_n \,\} \\ A(n) &= \sum_{I \in D_n} Pr(I) \cdot t(I) \end{aligned}$$

Im folgenden Abschnitt werden die vorgestellten Konzepte am Beispiel der linearen Suche praktisch angewandt und verdeutlicht.

C. Laufzeitkomplexitätsanalyse am Beispiel ‚Lineare Suche'

Gegeben sei der in Quellcode 1 implementierte Algorithmus zur Lösung der linearen Suche. Grundsätzlich ist die Analyse eines Algorithmus auch anhand einer Implementierungsskizze möglich.

Quellcode 1. Die Implementierung zur Lösung der linearen Suche

```
int linearSearch(int[] a, int length, int v) {
    for (int i = 0; i < length; i++) {
        if (a[i] == v) return i;
    }
    return -1;
}
```

Die Beschreibung des vorgestellten Algorithmus ist in Tabelle III dargestellt.

Tabelle III
IMPLEMENTIERUNGSCHARAKTERISTIKEN DER LINEAREN SUCHE

Problembeschreibung der linearen Suche	
Eingabe	Array `a` mit `length` vielen Einträgen sowie der gesuchte Wert `v`
Ausgabe	Index des gesuchten Wertes im Array, oder -1, wenn der Wert nicht enthalten ist
Elementare Operation	Vergleich zweier Zahlen konkret: `v` mit `a[i]`
Menge aller Eingaben	D_n ist die Menge aller Permutationen von n ganzen Zahlen. n entspricht dabei dem Wert von `length`

Mit obigen Angaben lässt sich die Laufzeitkomplexität sowohl für Best-, Worst- und Average-Case bestimmen. Im schlimmsten Fall steht die gesuchte Zahl an letzter Stelle oder ist gar nicht im Array enthalten; in jedem Fall müsste das komplette Array durchsucht werden. Damit ergibt sich für die Worst-Case-Komplexität $W(n) = n = \mathcal{O}(n)$. Im besten Fall steht das gesuchte Element an erster Stelle, wodurch nur ein einziger Vergleich nötig wäre, die Laufzeit also konstant ist, beschrieben durch $B(n) = 1 = \mathcal{O}(1)$. Die Bestimmung der durchschnittlichen Laufzeitkomplexität ist etwas aufwändiger, da hier die Wahrscheinlichkeiten für das Auftreten einer Eingabe berücksichtigt werden müssen. Für die lineare Suche lassen sich folgende zwei Szenarien festhalten:

1) Der gesuchte Wert `v` ist nicht in `a` enthalten.
2) Der gesuchte Wert `v` ist in `a` enthalten.

Jedes der beiden Szenarien hat eine entsprechende Average-Case-Laufzeit. Die durchschnittliche Zeitkomplexität ergibt sich aus der Kombination der beiden Laufzeiten wie folgt:

$$A(n) = Pr\{\texttt{v in a}\} \cdot A_{v \in a}(n) + Pr\{\texttt{v nicht in a}\} \cdot A_{v \notin a}(n)$$

Im Folgenden wird das Szenario ‚*v in a*' betrachtet. Es wird angenommen, dass alle Elemente im Array unterschiedlich sind und somit die Wahrscheinlichkeit, das gesuchte Element zu finden, bei jedem Vergleich gleich hoch ist, also $\frac{1}{n}$. Die benötigte Anzahl elementarer Operationen im Fall `v == a[i]` beträgt also $i+1$. Die durchschnittliche Laufzeit des Szenarios lässt sich nun wie folgt bestimmen:

$$\begin{aligned} A_{v \in a}(n) &= \sum_{i=0}^{n-1} Pr\{\texttt{v == a[i]} \mid \texttt{v in a}\} \cdot t(\texttt{v == a[i]}) \\ &= \sum_{i=0}^{n-1} \frac{1}{n} \cdot (i+1) \\ &= \frac{1}{n} \cdot \sum_{i=0}^{n-1} (i+1) \\ &= \frac{1}{n} \cdot \frac{n \cdot (n+1)}{2} \\ &= \frac{n+1}{2} \end{aligned}$$

Die Wahrscheinlichkeit für $Pr\{\texttt{v nicht in a}\}$ lässt sich durch $1 - Pr\{\texttt{v in a}\}$ beschreiben. Die durchschnittliche Laufzeit für das Szenario ‚*v nicht in a*' entspricht der Worst-Case-Komplexität, es gilt also $A_{v \notin a}(n) = n$. Damit lässt sich nun $A(n)$ wie folgt bestimmen:

$$\begin{aligned} A(n) &= Pr\{\texttt{v in a}\} \cdot \frac{n+1}{2} + Pr\{\texttt{v nicht in a}\} \cdot A_{v \notin a}(n) \\ &= Pr\{\texttt{v in a}\} \cdot \frac{n+1}{2} + (1 - Pr\{\texttt{v in a}\}) \cdot A_{v \notin a}(n) \\ &= Pr\{\texttt{v in a}\} \cdot \frac{n+1}{2} + (1 - Pr\{\texttt{v in a}\}) \cdot n \end{aligned}$$

Die Ergebnisse der Analyse sind nachfolgend in Tabelle IV dargestellt, wobei die Wahrscheinlichkeit des Vorkommens des gesuchten Wertes im Array berücksichtigt wird.

Tabelle IV
ERGEBNISSE DER AVERAGE-CASE-ZEITKOMPLEXITÄTSANALYSE AM BEISPIEL DER LINEAREN SUCHE

Wahrscheinlichkeit	**Average-Case-Zeitkomplexität**
$Pr\{\texttt{v in a}\} = 1$	$A(n) = \frac{n+1}{2}$, das heißt, etwa die Hälfte des Arrays wird durchsucht.
$Pr\{\texttt{v in a}\} = 0$	$A(n) = n = W(n)$, das heißt, das Array wird komplett durchsucht.
$Pr\{\texttt{v in a}\} = 0.5$	$A(n) = \frac{3n}{4} + \frac{1}{4}$, das heißt, etwa dreiviertel des Arrays wird durchlaufen.

In jedem der ermittelten Ergebnisse wird die Average-Case-Laufzeit durch $\mathcal{O}(n)$ beschränkt. Damit entspricht die durchschnittliche Zeitkomplexität der linearen Suche der Worst-Case-Komplexität, es gilt also $A(n) = W(n) \in \mathcal{O}(n)$.

Die Bestimmung der Average-Case-Zeitkomplexität ist üblicherweise aufwändiger als die anderen hier betrachteten Zeitkomplexitäten, jedoch hat diese Art der Zeitkomplexität eine hohe Bedeutung in der Praxis [8]. Die Gegenüberstellung der Zeitkomplexitäten kann beispielsweise Entwickler:innen helfen, Entscheidungen zu treffen, welcher Algorithmus für das Lösen eines Problems am geeignetsten ist. So hat beispielsweise der Sortieralgorithmus *Quicksort* eine durchschnittliche Zeitkomplexität von $\mathcal{O}(n \cdot log(n))$ kann im schlimmsten Fall aber eine Komplexität von $\mathcal{O}(n^2)$ erreichen. Für eine vollständige Komplexitätsanalyse des Quicksort-Algorithmus sei auf Moller [9] verwiesen. Die Arbeiten von Goldenreich [10] und Levin [11] ermöglichen einen noch tieferen Einblick in die in diesem Abschnitt betrachteten Konzepte der Laufzeitkomplexitätsanalyse.

III. KOMPLEXITÄTSKLASSEN

In Abschnitt II wurden bereits einige Komplexitätsklassen vorgestellt. In diesem Abschnitt wird nunmehr von konkreten Algorithmen abstrahiert und die Komplexität der zugrundeliegenden Probleme in den Vordergrund gerückt. Auf die wichtigsten Komplexitätsklassen wird in den folgenden Unterabschnitten eingegangen.

A. Die Klasse P

Die Komplexitätsklasse $\mathcal{P}$ enthält alle Probleme, die sich mit einer deterministischen Turingmaschine in polynomieller Zeit lösen lassen [5]. Dabei sei anzumerken, dass jedes real existierende Computerprogramm letztendlich auf eine deterministische Turingmaschine reduziert wird. Bei dieser Reduktion wird die Komplexitätsklasse nicht verlassen, das heißt, die Lösung von Problemen ist generell unabhängig von der Programmiersprache. Die in Abschnitt II-C vorgestellte *Lineare Suche* ist dieser Problemklasse zuzuordnen. Weitere in $\mathcal{P}$ enthaltene Probleme sind etwa das Sortierproblem, das Schaltkreis-Auswertungsproblem [12], aber auch Algorithmen in der Routenfindung, wie der bekannte Dijkstra-Algorithmus zum Finden des kürzesten Pfads zwischen zwei Knoten.

B. Die Klasse NP

Die Komplexitätsklasse $\mathcal{NP}$ enthält alle Probleme, die sich mit einer nichtdeterministischen Turingmaschine (NTM) in polynomieller Zeit lösen lassen [5]. Die NTM ist als Verallgemeinerung der deterministischen Turingmaschine anzusehen und hat die Möglichkeit in einer Situation zwischen mehreren Rechenschritten zu unterscheiden, wobei es sich hierbei nur um ein theoretisches Modell handelt [3]. Deterministische Turingmaschinen können Probleme, die in $\mathcal{NP}$ liegen, nur mit mindestens exponentieller Laufzeit lösen, aber in polynomieller Laufzeit überprüfen, ob ein gegebenes Ergebnis korrekt ist [13]. So ist etwa die Faktorisierung einer Zahl in ihre Primzahlen bis heute nicht in Polynomialzeit möglich, jedoch lässt sich in polynomieller Zeit überprüfen, ob ein gegebener Faktor tatsächlich ein Faktor der Zahl ist. Folglich sind in der Klasse $\mathcal{P}$ und $\mathcal{NP}$ alle entscheidbaren Probleme enthalten.

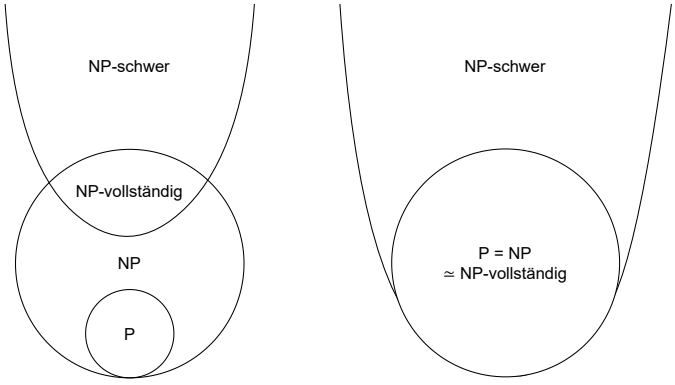
C. Die Klasse NP-schwer und NP-vollständig

Die Komplexitätsklasse $\mathcal{NP}$-schwer beinhaltet alle besonders schweren Probleme, die mindestens so schwer sind, wie jedes beliebige Problem aus $\mathcal{NP}$ [5]. Zusätzlich enthält $\mathcal{NP}$-schwer auch alle unlösbaren Probleme. Ein Problem gilt als $\mathcal{NP}$-schwer, wenn sich jedes Problem aus $\mathcal{NP}$ auf dieses Problem mittels Polynomialzeitreduktion reduzieren lässt [3], [5]. Probleme, die $\mathcal{NP}$-schwer sind und gleichzeitig in der Komplexitätsklasse $\mathcal{NP}$ enthalten sind, werden als $\mathcal{NP}$-vollständig bezeichnet. Ein Beispiel für ein $\mathcal{NP}$-schweres Problem, welches nicht in $\mathcal{NP}$ liegt, ist das *Halteproblem* für Turingmaschinen, welches nicht entscheidbar und damit nicht lösbar ist [13]. Das *Erfüllbarkeitsproblem der Aussagenlogik*, das *Traveling Salesman*-Problem und das *Rucksackproblem* sind klassische Beispiele für $\mathcal{NP}$-vollständige Probleme, welche gleichzeitig auch Beispiele für entscheidbare NP-schwere Probleme sind [5].

D. P-NP-Problem

Eine bis heute ungeklärte Frage ist, ob die zwei Komplexitätsklassen $\mathcal{P}$ und $\mathcal{NP}$ identisch oder verschieden sind. Bisher ist es noch nicht gelungen, einen Algorithmus zu finden, der ein beliebiges $\mathcal{NP}$-vollständiges Problem in polynomieller Laufzeit löst [13]. Gelänge es, einen solchen Algorithmus zu finden, dann wäre $\mathcal{P} = \mathcal{NP}$ gezeigt. Da ein solcher Algorithmus jedoch bis heute trotz aller Bemühungen nicht vorliegt, gehen die meisten Experten davon aus, dass $\mathcal{P} \neq \mathcal{NP}$ gilt [6]. Trotz mehrerer Beweisversuche konnte bis heute weder gezeigt werden, dass $\mathcal{P} = \mathcal{NP}$ gilt, noch dass $\mathcal{P} \neq \mathcal{NP}$ gilt [5], [13]. Das P-NP-Problem gehört zu den sieben Millennium-Problemen des Clay-Institutes, auf welches ein Preisgeld in Höhe von einer Million US-Dollar ausgesetzt ist [5], [14].

In Abbildung 1a sind die Enthaltenseinsbeziehungen für den Fall $\mathcal{P} \neq \mathcal{NP}$ dargestellt. Analog ist in Abbildung 1b der Fall $\mathcal{P} = \mathcal{NP}$ veranschaulicht.



(a) $\mathcal{P} \neq \mathcal{NP}$ (b) $\mathcal{P} = \mathcal{NP}$

Abbildung 1. Enthaltenseinsbeziehungen der Komplexitätsklassen P, NP, NP-schwer und NP-vollständig unter Betrachtung beidseitiger Ausgänge des P-NP-Problems

Bisher ist unbekannt, ob Probleme existieren, die weder in $\mathcal{P}$ noch in $\mathcal{NP}$-vollständig liegen. Der Satz von Ladner hat mithilfe eines künstlichen Problems bewiesen, dass es solche Probleme gibt, falls $\mathcal{P} \neq \mathcal{NP}$ gilt [15]. Die bereits erwähnte Primfaktorzerlegung sowie das *Graph isomorphism*-Problem [16] sind mögliche Kandidaten für Probleme, die zwar in $\mathcal{NP}$, aber nicht in $\mathcal{P}$ oder $\mathcal{NP}$-vollständig liegen.

Sollte es gelingen zu zeigen, dass $\mathcal{P} = \mathcal{NP}$ gilt, so wäre eine Lösung vieler praktischer Probleme aus $\mathcal{NP}$ in polynomieller Laufzeit möglich. Jedoch könnten die konstanten Faktoren sowie die Exponenten der Polynome dann so hoch sein, dass in der Praxis weiterhin bewährte Lösungsverfahren, wie etwa die in Abschnitt IV-B beschriebenen Approximationsalgorithmen verwendet werden [13].

Auf der anderen Seite ist der Beweis von $\mathcal{P} \neq \mathcal{NP}$ in einigen Bereichen sogar erwünscht. Beispielsweise gehen asymmetrische Verschlüsselungsverfahren davon aus, dass ihre Verschlüsselung eben nicht in Polynomialzeit gebrochen werden kann. Der Beweis von $\mathcal{P} = \mathcal{NP}$ ist generell, aufgrund der aktuellen Erwartungshaltung, mit weitaus größeren Konsequenzen verbunden.

Für weiterführende und tiefergehende Informationen sei auf die offizielle Problembeschreibung des P-NP-Problems verwiesen, welche von Stephen Cook [17] verfasst und publiziert wurde.

IV. Lösungsoptionen

Die in Abschnitt III dargestellten Komplexitätsklassen liefern eine Einordnung verschiedener Probleme in ihre Laufzeitkomplexität. Eine exakte Lösung für ein Problem in $\mathcal{NP}$ in akzeptabler Zeit zu berechnen, ist für große Problemgrößen nicht möglich.

Zumindest lässt sich für viele Probleme ein optimiertes Lösungsverfahren anwenden, wodurch das Problem für kleinere praxisrelevante Problemgrößen exakt gelöst werden kann. So sind auch Szenarien möglich, in denen ein Algorithmus mit exponentieller Laufzeit wie $\mathcal{O}(1.01^n)$ für kleine Eingabegrößen schneller ein exaktes Ergebnis errechnet, als ein Algorithmus mit der Polynomiallaufzeit $\mathcal{O}(n^4)$ [18]. Für viele bekannte $\mathcal{NP}$-Probleme lassen sich exakte Lösungen für das Problem durch Methoden der ganzzahlig linearen Optimierungen, insbesondere Branch-and-Cut, für praktisch relevante Größenordnungen beweisbar optimal berechnen [19]. Die Optimierungsmöglichkeiten für exakte Lösungen für $\mathcal{NP}$-Probleme sind sehr problemspezifisch und werden deshalb in dieser Arbeit nicht näher behandelt [1].

Trotz der Optimierungsmöglichkeiten, bleibt die Laufzeit für exakte Lösungen bei $\mathcal{NP}$-Problemen exponentiell. Ein anderer Ansatz, um Probleme in $\mathcal{NP}$ zu lösen, sind Annäherungen an die exakte Lösung, die in Polynomialzeit berechnet werden. Dabei wird zwischen Heuristiken und Approximationsalgorithmen unterschieden. Die beiden Ansätze und dessen Unterschiede werden in den folgenden Abschnitten erläutert.

A. Heuristiken

Heuristiken sind eine Vorgehensweise zur Lösung von mathematischen Problemen, wobei die Lösungsmethode auf der Basis von Erfahrung oder Urteilsvermögen meist zu guten

Lösungen führt, die aber nicht notwendigerweise optimal ist. Häufig agieren Heuristiken nach einem *greedy* Auswahlverfahren. Hierbei wird die momentane Gewinnmaximierung betrachtet, sodass die Gefahr besteht, in lokale Optima zu verfallen [20].

Ein anschauliches Beispiel hierfür ist das bekannte Traveling Salesman Problem (TSP). Die Problemstellung liegt darin, in einem Graphen mit einer beliebigen Menge von Knoten und Kanten, von einem Startknoten beginnend, eine minimale Rundreise durchzuführen, bei der jeder Knoten genau einmal erreicht wird. Die Kantenlängen definieren die Distanz zwischen zwei Knoten. Das symmetrische TSP beschreibt die zusätzliche Eigenschaft, dass die Distanz zwischen zwei Knoten in beide Richtungen gleich lang ist. Eine einfache und schnelle Heuristik, die dieses Problem löst, ist die Nächste-Nachbar-Heuristik. Diese wählt von jedem Knoten ausgehend, immer den Knoten, der die geringste Distanz zu dem aktuellen Knoten hat. Dies führt oft zu guten Ergebnissen, die in vielen Fällen nicht weit von der Optimallösung abweichen, jedoch kann keine Güte garantiert werden. Dieses Vorgehen hat die Laufzeitkomplexität $\mathcal{O}(n^2)$. Dies ist ein enormer Unterschied zur exponentiellen Laufzeit, die für eine exakte Lösung benötigt wird [20].

Neben der Nearest-Neighbor-Heuristik, die der Klasse der Eröffnungsheusristiken zugehörig ist, kommen häufig auch Einfügeheuristiken zum Einsatz. Diese starten mit einem kleinem Ausschnitt des Gesamtgraphen, für den sich eine exakte Lösung noch in kurzer Laufzeit berechnen lässt und fügen dann die übrigen Knoten nach verschiedenen Vorgehensweisen Schritt für Schritt in den Graphen ein, bis die gesamte Rundreise komplett ist. Häufig wird nachdem eine Lösung durch eine Heuristik berechnet wird, eine Verbesserungsheuristik angewendet, um die Lösung zu optimieren. Eine Kombination aus exakten Lösungen und verschiedenen Heuristiken, die nacheinander angewendet werden, ist eine häufige Vorgehensweise, um exakte Lösungen gut anzunähern. Hierbei ist zu beachten, dass das Prinzip Teile-und-Herrsche nicht für NP-schwere Probleme angewendet werden kann, um eine exakte Lösung zu errechnen, weil die Zusammensetzung optimaler Teillösungen für NP-Probleme nicht in polynomieller Laufzeit möglich ist [20].

B. Approximationsalgorithmen

Ein weiterer Lösungsansatz, um $\mathcal{NP}$-Probleme anzunähern, sind Approximationsalgorithmen, dessen Lösung eine beweisbare Güte aufweist. Dies ist ein fundamentaler Unterschied zu Heuristiken, dessen Lösungen keine quantifizierbare Güte garantieren [20].

Wanka [21] unterscheidet zwischen 3 verschiedenen Arten von Approximationen und einhergehenden Gütegarantien. Diese werden im Folgenden näher erläutert.

1) *Approximationen mit absoluter Gütegarantie*
   Bezeichne Π ein Optimierungsproblem für ein Minimierungs- oder Maximierungsproblem und *I* eine Instanz von Π. *A* sei ein Approximationsalgorithmus für Π. *A(I)* wird als *absoluter Approximationsalgorithmus* bezeichnet, falls ein $K \in \mathbb{N}_0$ existiert, sodass gilt:

   $|$ *A(I)*-OPT(*I*) $| \leq K$, wobei:

   A(I) : Wert der Lösung, die *A* für *I* berechnet.

   $OPT(I)$: Wert einer optimalen Lösung für *I*.

   Diese Art von Approximation, die einen konstanten absoluten Wert als maximale Abweichung von der Optimallösung beschreibt, kommt bei Problemen in $\mathcal{NP}$ nur sehr selten vor. Für das Problem der Graphenfärbbarkeit existiert eine solche Approximation [21].
2) *Approximationen mit relativer Gütegarantie*
   Ein Algorithmus wird als relativer Approximationsalgorithmus bezeichnet, wenn er für ein vorgegebenes Optimierungsproblem Π für jedes *A(I)* von Π einen Wert *A(I)* liefert mit

   $R_A(I) \leq K$, wobei:

   $$R_A(I) = \begin{cases} \frac{A(I)}{OPT(I)} & \text{falls } \Pi \text{ Minimierungsproblem} \\ \\ \frac{OPT(I)}{A(I)} & \text{falls } \Pi \text{ Maximierungsproblem} \end{cases}$$

   K : Konstante ≥ 1

   Der Wert von $R_A(I)$ wird auch als Approximationsfaktor bezeichnet. Ein relativer Approximationsalgrithmus ist für viele Probleme in $\mathcal{NP}$ bekannt. Das Rucksack-Problem (*auch knapsack-problem*) ist ein bekanntes Optimierungsproblem und behandelt folgende Problematik. Eine Menge von Objekten ist durch ein Gewicht und einen Nutzwert definiert. Daraus soll eine Teilmenge ausgewählt werden, die ein vorgeschriebenes Maximalgewicht nicht überschreitet und dessen summierter Nutzwert maximal ist. Für das Rucksack-Problem existiert ein Approximationsalgorithmus mit Approximationsfaktor 1,5. Das bedeutet, dass das durch den Approximationsalgorithmus errechnete Ergebnis auf jeden Fall maximal so groß ist, wie *1,5 * Optimallösung* [21]. Für andere Probleme wie das MAX-3-SAT-Problem, ist die Nichtapproximierbarkeit durch einen Approximationsfaktor bewiesen [22].
   Nachfolgend wird das Vertex-Cover-Problem betrachtet und für einen Approximationsalgorithmus, der das Problem löst, ein Approximationsfaktor von 2 bewiesen. Dieses Problem wurde aufgrund der anschaulichen Darstellungsmöglichkeit und der geringen Komplexität der bestmöglichen Approximation, ausgewählt. Das Vertex-Cover-Problem liegt in $\mathcal{NP}$-vollständig und ist somit nicht durch eine deterministische Turingmaschine in Polynomialzeit lösbar, weshalb eine Approximation in Polynomialzeit hilfreich sein kann, um schnell ein verlässliches Ergebnis zu erhalten [23]. Das Vertex-Cover-Problem ist folgendermaßen definiert:
   Sei $G = (N, E)$ ein ungerichteter Graph mit der Knotenmenge *N* und der Kantenmenge E. Dann ist eine Teilemenge $U \subseteq N$ eine Knotenüberdeckung (Vertex Cover) von G, wenn jede Kante von G wenigstens einen Knoten

aus U enthält. Eine Knotenüberdeckung U von G nennt man minimal, wenn es keinen Knoten $n \in U$ gibt, sodass U ohne n immer noch eine Knotenüberdeckung ist. Die Anzahl der Knoten einer kleinsten Knotenüberdeckung von G nennt man Knotenüberdeckungszahl von G [24].

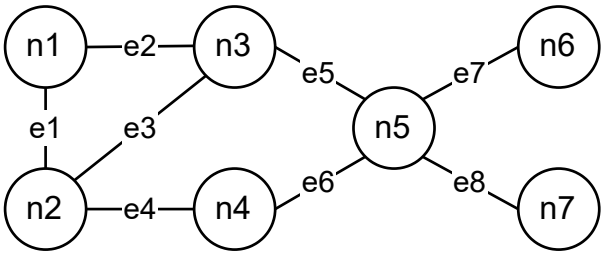


Abbildung 2. Beispielgraph Vertex Cover

Abbildung 2 zeigt einen beispielhaften Graphen für das Problem mit den Knoten $n1 - n7$ und den Kanten $e1 - e8$. Anhand dieses Graphen wird die Approximation beispielhaft näher erläutert. Die Knotenüberdeckungszahl dieses Graphen beträgt 3 ($\{n2, n3, n5\}$). Der Beweis für die Güte der Approximation eines Algorithmus, der das Vertex-Cover-Problem löst, lässt sich in drei Phasen einteilen:

a) *Algorithmus aufstellen*
Als Erstes wird ein Algorithmus betrachtet, der das Vertex-Cover-Problem löst. Algorithmus 1 beschreibt ein mögliches Lösungsverfahren und wird im Folgenden erläutert. Der Algorithmus erhält als Eingabe einen Graphen *G*, bestehend aus der Knotenmenge N und der Kantenmenge E. In Zeile 1 wird eine Ergebnismenge C (Cover) definiert. Die Zeilen 3-6 werden solange ausgeführt, bis die Kantenmenge E leer ist. Dabei wird zuerst eine beliebige Kante e aus der Menge aller Kanten gewählt (Zeile 3), dann die Endknoten $n1$ und $n2$ der selektierten Kante e der Ergebnismenge C angehängt (Zeile 4 und 5) und anschließend alle Kanten aus E entfernt, die inzident zu $n1$ oder $n2$ sind. Ist die Abbruchbedingung erfüllt, wird wie in Zeile 8 dargestellt die Ergebnismenge C zurückgegeben.
Diese Ergebnismenge enthält die Menge der Knoten, dessen Kanten alle im Graphen enthaltenen Kanten abdecken, wodurch das Vertex-Cover-Problem gelöst ist. Eine mögliche Lösungsmenge für den Graphen in Abb. 2, den der Algorithmus 1 errechnet, wäre $\{n1, n2, n3, n5\}$.

b) Untere Schranke definieren
Im ersten Schritt wurde ein Algorithmus definiert, der das Vertex-Cover-Problem definitiv löst. Jedoch ist noch unklar, wie gut dieser Algorithmus verglichen mit der Optimallösung ist. Um die Güte zu beweisen, muss zuerst eine untere Schranke für das Problem definiert werden. Eine untere Schranke ist folgendermaßen definiert: Ein Element $b \in M$ heißt untere Schranke von T, wenn $b \leq x$ für alle $x \in T$ gilt.

Algorithmus 1 : approx_vertex_cover(G) [25]

```
1 C ← ∅
2 while E ≠ ∅ do
3 |   wähle beliebige Kante e ∈ E
4 |   n1, n1 ← Endknoten der Kante e
5 |   C ← C ∪ {n1, n2}
6 |   entferne alle Kanten aus E, die inzident zu n1 oder
  |    n2 sind
7 end
8 return C
```

Das Ziel ist hierbei, eine untere Schranke zu definieren, die möglichst nah an der Optimallösung ist. Um für das Vertex-Cover-Problem eine gute untere Schranke zu definieren, müssen die disjunkten Kanten betrachtet werden. Zwei Kanten sind disjunkt zueinander, wenn sie sich keinen Endknoten teilen. In Abb. 2 sind beispielsweise die Kanten $e2$ und $e4$ disjunkt zueinander, $e5$ und $e6$ aber nicht, da sie sich den gemeinsamen Endknoten $n5$ teilen.
Die untere Schranke für das Vertex-Cover-Problem ist definiert durch $\mid E^* \mid \leq OPT(G)$, wobei:

E ← Alle Kanten im Graph *G*

E* $\subseteq E$ ← Beliebige Menge disjunkter Kanten in *G*

OPT(G) ← Minimale Knotenüberdeckungszahl für *G*

Diese Erkenntnis ergibt sich, weil bei einer Menge von disjunkten Kanten im Graphen immer mindestens ein Endknoten jeder disjunkten Kante in der Lösungsmenge enthalten sein muss. Ansonsten wäre die Kante durch keinen anderen Knoten erreichbar, da die Kanten alle disjunkt zueinander sind. Somit ist die Kardinalität einer beliebigen Menge von disjunkten Kanten im Graph, die minimal mögliche Optimallösung für diesen Graphen [25].

c) Untere Schranke mit Algorithmus verknüpfen
Um die Güte für den Algorithmus 1 zu garantieren, muss die untere Schranke mit der Lösung des Algorithmus in Beziehung gesetzt werden. Dies ist durch folgende Erkenntnisse möglich.

Zeile 3 in Algorithmus 1 wählt immer nur disjunkte Kanten. Dies ergibt sich, weil in Zeile 6 alle Kanten, die inzident zu den Endknoten der zufällig gewählten Kante sind, entfernt werden.

In Zeile 4 werden die Endknoten der Kante gewählt und in Zeile 5 der Ergebnismenge hinzugefügt. Somit ergibt sich, dass die Kardinalität der Ergebnismenge des Algorithmus zweimal der Anzahl der selektierten Kanten entspricht. Somit entspricht $\mid approx\_vertex\_cover(G) \mid$ = *2 * Anzahl selektierter Kanten (disjunkt)*.

Durch die Erkenntnis, dass die Anzahl der disjunkten Kanten einer beliebigen Menge im Graphen die untere Schranke für die minimale Knotenüberde-

ckung darstellt, ergibt sich folgende Gütegarantie für Algorithmus 1:

$$| \textit{approx\_vertex\_cover(G)} | \leq 2 * \text{OPT(G)}$$

Ein Approximationsfaktor von 2 garantiert nun, dass für einen beliebigen Graphen durch den Algorithmus 1 eine Lösungsmenge errechnet werden kann, dessen Kardinialität sicher nicht größer ist, als optimale Knotenüberdeckungszahl * 2. Obwohl dies keine starke Garantie ist und obwohl der beschriebene Algorithmus trivial ist, stellt dieser Algorithmus den aktuell besten bekannten Approximationsalgorithmus dar, der das Problem in Polynomialzeit löst. Trotz dieser Eigenschaft, wird der Algorithmus in der Praxis nicht verwendet. Dies liegt an der schwachen Garantie und an dem schlechten durchschnittlichen Ergebnis für Graphen-Formationen, die häufig in der Praxis auftreten. Stattdessen werden in der Praxis Heuristiken verwendet, die das Problem durchschnittlich besser lösen [25].

3) Approximationsschemata
Die beschriebenen Ansätze der relativen und absoluten Gütegarantie geben eine feste Abweichung zur optimalen Lösung an. Sie bieten keine Möglichkeit noch bessere Lösungen zu errechnen, außer durch die exakte Lösung des Problem mit exponentieller Laufzeit. Approximationsschemata hingegen bieten die Möglichkeit, den relativen Fehler als zusätzlichen Eingabeparameter mit zu definieren, sodass die Ausgabe höchstens um diesen Fehler von der Optimallösung abweicht. Die Laufzeit hängt hierbei jedoch von diesem relativen Fehler ab. Ziel ist es, dass das Verhältnis von Laufzeit zu Optimumabweichung in einem guten Verhältnis steht. Für das Rucksack-Problem existiert ein solches Approximationsschemata [21].

Dieser Abschnitt hat grundlegende Lösungsoptionen für schwere Probleme betrachtet und dabei Annäherungsmöglichkeiten durch Heuristiken und Approximationsalgorithmen dargestellt. Dabei wurden 3 unterschiedliche Approximationsarten vorgestellt und die Approximation mit relativer Gütegarantie am Beispiel des Vertex-Cover-Problems ausführlich beschrieben. Eine mögliche Praxisanwendung der Problemstellug wäre die Platzierung von Kameras an Kreuzungen von Straßen. Hierbei würden Kreuzungen den Knoten und die Straßen den Kanten entsprechen. Ziel wäre es, die minimale Anzahl an Kameras zu verwenden, um alle Straßen in einem bestimmten Bereich abzudecken, um Kosten zu sparen [26]. Der nächste Abschnitt zeigt durch die Möglichkeiten des Quantencomputing ein anderes Modell zur Lösung von schweren Problemen vor.

V. Auswirkungen des Quantencomputings

Die klassischen Rechner nutzen Bits, die entweder eine 0 oder eine 1 als Zustand annehmen können. Ein grundlegend anderes Konzept liegt bei Quantencomputern vor. Diese rechnen nicht mit Bits, sondern mit Quantenbits, die anders als klassische Rechner zu einem Zeitpunkt nicht nur einen Zustand, sondern mehrere Zustände gleichzeitig haben können. Somit können auf einem Quantencomputer mehrere Rechenschritte parallel, anstatt sequentiell durchgeführt. Dieser Zustand wird Superposition genannt. Ein klassischer Rechner kann also mit *n* Bits 2^n verschiedene Zahlen darstellen, zu jedem Zeitpunkt aber nur eine davon speichern. Ein Quantencomputer hingegen kann mit ebenso vielen Quantenbits 2^n Zahlen *gleichzeitig* darstellen [27].

Noch deutlicher werden die Unterschiede zwischen klassischen Rechnern und Quantencomputern durch ein Forschungsprojekt, das 2019 von Google durchgeführt wurde. Hierbei wurde ein Problem durch einen Quantencomputer mit 53 Qubits in 200 Sekunden gelöst, das mit einem klassischen Rechner 10.000 Jahre gedauert hätte [28]. Obwohl das Problem explizit auf die Funktionsweise des Quantencomputers zugeschnitten wurde und weitere Kritik geäußert wurde, verdeutlichen Experiemente wie diese den enormen Performancevorteil von Quantencomputern gegenüber klassischen Rechnern. Nichtsdestotrotz sind universell programmierbare Quantencomputer noch in unbekannter Ferne [29].

Durch die theoretisch definierte Funktionsweise von Quantencomputern, können diese mit den Terminologien der theoretischen Informatik in Bezug gestellt werden. Im Folgenden werden die aktuellen Erkenntnisse der möglichen Auswirkungen von Quantcomputern auf die Komplexitätstheorie aufgeführt. Dies soll lediglich als Ausblick in dieses Themenfeld dienen.

1) *Die Menge der lösbaren Probleme für einen Quantencomputer ist nicht größer als für einen klassischen Computer.* Dies ergibt sich daraus, dass ein klassischer Computer das Verhalten eines Quantencomputers in sequentieller Abfolge simulieren kann. Somit kann jedes Problem, das von einem Quantencomputer gelöst werden kann auch von einem klassischen Rechner gelöst werden. Ein untentscheidbares Problem wie das Halteproblem, das die Problemstellung behandelt, ob ein Programm terminiert oder nicht, bleibt auch für einen Quantencomputer unentscheidbar [30].
2) *Einige Probleme in $\mathcal{NP}$-schwer lassen sich (theoretisch) durch Quantencomputer in Polynomialzeit lösen.* Dies zeigt der Shor-Algorithmus, wodurch auf einem Quantencomputer ein nichttrivialer Teiler einer zusammengesetzten Zahl in Polynomialzeit berechnet werden kann [31]. Dieses Problem kann von klassischen Computern nur in exponentieller Laufzeit gelöst werden. Der Algorithmus würde aktuelle Kryptographiesysteme aufbrechen, jedoch existiert aktuell kein Quantencomputer, bei dem die Fehlerrate gering genug ist, um den Algorithmus praktisch anzuwenden.
3) *Es ist unklar, ob ein Quantencomputer ein $\mathcal{NP}$-vollständiges Problem in Polynomialzeit lösen kann.* Bis heute ist kein bewiesener Ansatz bekannt, um durch einen Quantencomputer ein Problem aus $\mathcal{NP}$-vollständig in Polynomialzeit zu lösen. Dies hätte die Auswirkung, dass jedes Problem aus $\mathcal{NP}$ durch einen Quantencomputer in Polynomialzeit lösbar wäre, da sich jedes Problem aus

$\mathcal{NP}$ auf jedes Problem aus $\mathcal{NP}$-vollständig reduzieren lässt. Die Tendenz vieler Experten ist, dass ein Quantencomputer nicht in der Lage ist, ein $\mathcal{NP}$-vollständiges Problem in Polynomialzeit zu lösen [29].

4) *Selbst wenn $\mathcal{NP}$=P bewiesen wird, gibt es Probleme, die von Quantencomputern effizient gelöst werden können, von klassischen Rechnern aber nicht.* Für Quantencomputer wurden eigene Komplexitätsklassen definiert. Durch einen hergestellten Bezug der Quantencomputer-Komplexitätsklassen zu den klassischen Komplexitätsklassen wurde für einige Probleme aufgezeigt, dass diese für Quantencomputer effizient lösbar sind, für klassische Rechner jedoch nicht. Der Groover-Algorithmus ist ein Quantenalgorithmus, der zur Suche in einer unsortierten Datenbank genutzt werden kann und bei n Datenbankeinträgen eine Laufzeitkomplexität von $\mathcal{O}(\sqrt{n})$ aufweist. Dies kann für große Problemordnungen ein riesiger Unterschied zu der Lautzeitkomplexität $\mathcal{O}(n)$ sein, die für klassische Rechner gilt [32].

VI. Fazit

In dieser Arbeit wurden wesentliche Aspekte der Komplexitätstheorie als Teilgebiet der theoretischen Informatik beleuchtet. Die Komplexitätstheorie ist ein sehr stabiles und gut erforschtes Gebiet und hat eine sehr hohe praktische Relevanz.

Die O-Notation bildet die Grundlage zur Beurteilung von Zeit- und Platzkomplexitäten von Algorithmen. Dabei wird die Komplexität eines Algorithmus nach oben abgeschätzt, wodurch sich bereits konkrete Komplexitätsklassen bilden lassen. Die Laufzeitkomplexitäten von Algorithmen lassen sich noch genauer durch die Differenzierung von Worst-, Best- und Average-Case-Analysen beschreiben, wobei insbesondere die Average-Case-Zeitanalyse aufgrund der Berücksichtigung von Verteilungen mit deutlich mehr Aufwand verbunden ist.

Im Gegensatz zur O-Notation bilden die Komplexitätsklassen $\mathcal{P}$ und $\mathcal{NP}$ eine breitere Klassifizierung von Problemen und ihren Algorithmen. Die Klasse $\mathcal{P}$ enthält alle Probleme, die eine deterministische Turingmaschine in polynomieller Zeit lösen kann, wohingegen die Klasse $\mathcal{NP}$ alle Probleme enthält, die von einer deterministischen Turingmaschine nur in exponentieller Laufzeit gelöst werden kann. Ob diese beiden Klassen identisch oder verschieden sind, ist nach wie vor eine der großen ungelösten Fragen der Komplexitätstheorie. Die Wichtigkeit dieses sogenannten P-NP-Problems wird durch die sowohl theoretische als auch praktische Relevanz unterstrichen.

Aufgrund der hohen Praxisrelevanz von vielen Problemen in $\mathcal{NP}$, werden Lösungen für diese Probleme benötigt, die nicht nur für kleine Problemgrößen berechnet werden können. Dafür können Heuristiken verwendet werden, die das Problem nach einem methodischen Verfahren, das häufig gute Ergebnisse liefert, in Polynomiallaufzeit lösen. Jedoch kann hierbei keine Aussage über die Güte der Lösung getroffen werden. Die durch die Heuristik errechnete Lösung kann beliebig schlecht werden. Um eine Gütegarantie für die angenäherte Lösung zu erhalten, können Approximationsalgorithmen verwendet werden, die mathematisch bewiesen für jede Eingabe eine festgelegte Maximalabweichung zur Optimallösung aufweisen. Das Beispiel des Vertex-Cover-Problems hat verdeutlicht, dass bereits für sehr triviale Algorithmen Gütegarantien existieren können.

Quantencomputer haben durch ihre grundlegend andere Funktionsweise einen enormen Performancevorteil gegenüber klassischen Rechnern. Durch die Quantenbits lassen sich Zustände gleichzeitig abbilden und somit Abläufe parallelisieren. Die Arbeit hat Auskunft über die aktuellen bekannten Auswirkungen der Quantencomputer auf die Komplexitätstheorie gegeben. Obwohl gravierende Vorteile im Vergleich zu klassischen Rechnern in Bezug auf die Komplexität einiger Probleme bewiesen sind, existieren in der Praxis noch keine Quantencomputer, die die theoretischen Konzepte umsetzen. Die Aufrechterhaltung des Quantenzustands und die Minimierung der Fehlerquote sind große Herausforderungen, die nicht gelöst sind daher universell programmierbare Quantencomputer noch in unbekannter Ferne sind.

Die Arbeit hat generelle Konzepte der Komplexitätstheorie aufbereitet, durch Beispiele verdeutlicht, neuste Entwicklungen in ungeklärten Fragen präsentiert und einen Ausblick auf die Auswirkungen des Quantencomputings gegeben.

Literatur

[1] J. Hromkovic, *Komplexitätstheorie*. Wiesbaden: Teubner, 2007, pp. 206–261. [Online]. Available: https://doi.org/10.1007/978-3-8351-9115-0_6

[2] J. Köbler and O. Beyersdorff, *Von der Turingmaschine zum Quantencomputer — ein Gang durch die Geschichte der Komplexitätstheorie*, 09 2006, pp. 165–195.

[3] L. Priese and K. Erk, *Komplexität*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2018, pp. 453–486. [Online]. Available: https://doi.org/10.1007/978-3-662-57409-6_15

[4] D. Sabel, "Kapitel 10: Komplexität von algorithmen und sortierverfahren," https://www.sosy-lab.org/Teaching/2018-WS-InfoEinf/vorlesungsfolien/10-komplexitaet-4s.pdf, letzter Zugriff 05.02.2022.

[5] G. Vossen and K.-U. Witt, *Komplexität*. Wiesbaden: Springer Fachmedien Wiesbaden, 2016, pp. 383–421. [Online]. Available: https://doi.org/10.1007/978-3-8348-2202-4_11

[6] B. Nebel, "Theoretische informatik, woche 11: Komplexitätstheorie - einführung," https://videoportal.uni-freiburg.de/video/Theoretische-Informatik-Woche-11-Komplexitaetstheorie-Einfuehrung/378d716e45ebf3da1046e61f2a30a8bf, letzter Zugriff 05.02.2022.

[7] T. Guenther, "Laufzeit und komplexität," https://www.elaspix.de/Lehre/Java2/LaufzeitSortierenRekursion/Laufzeit.pdf, 2015, letzter Zugriff 05.02.2022.

[8] J.-P. Katoen, "Datenstrukturen und Algorithmen," 2015, letzter Zugriff 05.02.2022.

[9] F. Moller, "Analysis of quicksort," https://www.cs.swan.ac.uk/~csfm/Courses/CS_332/quicksort.pdf, 2017, letzter Zugriff 05.02.2022.

[10] O. Goldenreich, *Introduction to Complexity - Lecture Notes*. Department of Computer Science and Applied Mathematics, 1999. [Online]. Available: https://www.wisdom.weizmann.ac.il/~oded/PS/CC/all.pdf

[11] L. A. Levin, "Average case complete problems," *SIAM Journal on Computing*, vol. 15, no. 1, pp. 285–286, 1986. [Online]. Available: https://doi.org/10.1137/0215020

[12] L. M. Goldschlager, "The monotone and planar circuit value problems are log space complete for p," *SIGACT News*, vol. 9, no. 2, p. 25–29, jul 1977. [Online]. Available: https://doi.org/10.1145/1008354.1008356

[13] L. Fortnow, "The status of the p versus np problem," *Commun. ACM*, vol. 52, no. 9, p. 78–86, sep 2009. [Online]. Available: https://doi.org/10.1145/1562164.1562186

[14] C. M. Institute, "P vs np problem," letzter Zugriff: 06.02.2022. [Online]. Available: http://www.claymath.org/millennium-problems/p-vs-np-problem

[15] R. E. Ladner, “On the structure of polynomial time reducibility,” *J. ACM*, vol. 22, no. 1, p. 155–171, jan 1975. [Online]. Available: https://doi.org/10.1145/321864.321877
[16] L. Babai, “Graph isomorphism in quasipolynomial time,” 2016.
[17] S. Cook, “The p versus np problem,” letzter Zugriff: 06.02.2022. [Online]. Available: http://www.claymath.org/sites/default/files/pvsnp.pdf
[18] G. J. Woeginger, “Exact algorithms for np-hard problems: A survey,” in *Combinatorial Optimization*, 2001.
[19] J. E. Mitchell, “Branch-and-cut algorithms for combinatorial optimization problems,” in *Branch-and-Cut Algorithms for Combinatorial Optimization Problems*, 1988.
[20] R. Mart, P. M. Pardalos, and M. G. C. Resende, *Handbook of Heuristics*, 1st ed. Springer Publishing Company, Incorporated, 2018.
[21] R. Wanka, *Approximationsalgorithmen*. Springer Fachmedien Wiesbaden, 2006.
[22] M. M. Klaus Jansen, *Approximative Algorithmen und Nichtapproximierbarkeit (De Gruyter Lehrbuch)*, 1st ed., ser. De Gruyter Lehrbuch. De Gruyter, 2008.
[23] “Proof that vertex cover is np complete,” https://www.geeksforgeeks.org/proof-that-vertex-cover-is-np-complete/, 2018.
[24] R. Diestel, *Graphentheorie*, 3rd ed., ser. Springer-Lehrbuch Masterclass. Springer, 2006. [Online]. Available: libgen.li/file.php?md5=664569d5f6f75c54803cac5d42a817b9
[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press, 2009. [Online]. Available: libgen.li/file.php?md5=47214322fa81f9292af442c9cae67335
[26] T. Gopal, G. Jäger, and S. Steila, *Theory and Applications of Models of Computation: 14th Annual Conference, TAMC 2017, Bern, Switzerland, April 20-22, 2017, Proceedings*. Springer, 2017, vol. 10185.
[27] M. Homeister, *Quantum Computer verstehen*. Springer Fachmedien Wiesbaden, 2018.
[28] F. Arute, K. Arya, and R. Babbush, “Quantum supremacy using a programmable superconducting processor,” *Nature*, vol. 574, no. 7779, pp. 505–510, Oct 2019. [Online]. Available: https://doi.org/10.1038/s41586-019-1666-5
[29] S. Aaronson, “The limits of quantum,” *Scientific American*, vol. 298, no. 3, pp. 62–69, 2008. [Online]. Available: http://www.jstor.org/stable/26000518
[30] N. Linden and S. Popescu, “The halting problem for quantum computers,” 1998.
[31] P. W. Shor, “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer,” *SIAM Journal on Computing*, vol. 26, no. 5, p. 1484–1509, Oct 1997. [Online]. Available: http://dx.doi.org/10.1137/S0097539795293172
[32] L. K. Grover, “A fast quantum mechanical algorithm for database search,” in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, ser. STOC ’96. New York, NY, USA: Association for Computing Machinery, 1996, p. 212–219. [Online]. Available: https://doi.org/10.1145/237814.237866