universität
wien

1. BACHELORARBEIT

Titel der Bachelorarbeit

Einführung in Komplexitätsklassen

Verfasser
Franz Schekolin

angestrebter akademischer Grad
Bachelor of Education (BEd)

Wien, im Februar 2021

Studienkennzahl lt. Studienblatt:	UA 198 411 420 02
Studienrichtung lt. Studienblatt:	UF Mathematik
Betreuer:	Vera Fischer, Privatdoz. PhD

Abstract

Die Komplexitäts-Theorie beschäftigt sich mit der Berechnung von Problemen durch Computer. Die Gruppierung dieser Probleme anhand ihrer Schwierigkeit ist der Grundgedanke von Komplexitätsklassen. Dabei ist die Frage, welche Probleme denn überhaupt von einem Computer berechenbar sind, zentral. Einige Beispiele von berechenbaren und nicht-berechenbaren Problemen werden in dieser Arbeit aufgezeigt.

Bevor die Komplexität von Problemen festgestellt werden kann, ist es notwendig, einige Grundbegriffe zu definieren. Zu Beginn ist eine genaue Definition eines mathematischen Modells zur Berechnung von Problemen wesentlich. Dieses Modell soll einerseits so einfach wie möglich und andererseits genauso mächtig wie ein realer Computer sein. Das berühmteste Modell hierfür ist die sogenannte Turing-Maschine. Um diese erarbeiten zu können, betrachten wir zunächst die Automaten-Theorie, da die Turing-Maschine darauf aufbaut. Mit Beispielen werden die Definitionen nicht nur theoretisch, sondern auch praktisch bearbeitet.

Mit der Turing-Maschine wird ein Modell eingeführt, das alle Berechnungen durchführen kann, die auch von modernen Computern berechnet werden können. Die große Frage, die sich stellt, ist: Wie viel Zeit und wie viel Speicherplatz wird zur Problemlösung benötigt? Wie wichtig die Einteilung von Problemen in ihre benötigte Zeit ist, und wie man diese Kategorisierung vornehmen kann, ist die Hauptaufgabe der Komplexitäts-Theorie. Die Definition dreier grundlegender Komplexitätsklassen, die Probleme in ihre für die Berechnung benötigte Zeit einteilen sowie das Erkennen der Wichtigkeit dieser Kategorisierung ist das Ziel dieser Arbeit.

Inhaltsverzeichnis

Die folgenden Kapitel basieren inhaltlich auf den Büchern *Computational complexity* [4], *Introduction to the theory of computation* [5] und *Computational Complexity: A Modern Approach* [1]. Direkte Zitate und Definitionen werden extra angeführt.

1 Automaten-Theorie

In diesem Kapitel werden grundlegende Computer-Modelle betrachtet. Wir beschäftigen uns hauptsächlich mit dem sogenannten ***endlichen Automaten***. Dieser gibt eine leicht verständliche Einführung in das Thema und beinhaltet alle Grundlagen, die für das Verständnis einer Turing-Maschine gebraucht werden.

1.1 Endliche Automaten

1.1.1 Einführende Beispiele

Wie der Name schon verrät, handelt es sich hierbei um ein Modell, das nur eine beschränkte Menge an Input verarbeiten kann. Trotzdem ist dieses Modell sehr hilfreich, da oft gar nicht viel Input benötigt wird, um die gewünschte Funktion zu gewährleisten. In unserem täglichen Leben benutzen wir oft unbewusst elektronische Geräte, die auf dem Modell eines endlichen Automaten basieren. Ein elektrisches Eingangstor in einem Supermarkt ist zum Beispiel so ein Gerät.

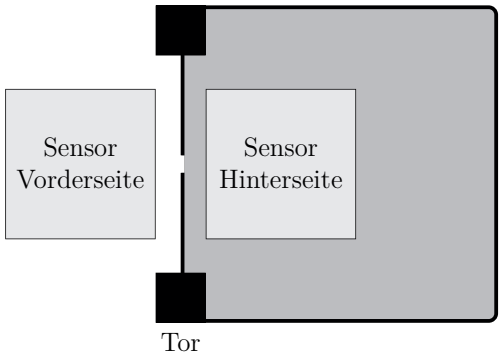

Abbildung 1: Vogelperspektive auf das automatische Tor

Wir betrachten ein Tor, das beim Öffnen nach innen schwenkt. Vor und nach dem Tor sind, wie in Abbildung 1 gezeigt, Sensoren angebracht, die erkennen ob jemand auf der Vorder- bzw. Hinterseite des Tors steht. Der Zustand des Tors kann entweder „OFFEN“ oder „ZU“ sein. Es gibt nun vier mögliche Inputs: „VORNE“ (es steht jemand auf dem Sensor der Vorderseite), „HINTEN“ (es steht jemand auf dem Sensor der Hinterseite), „BEIDE“ (auf beiden Sensoren steht jemand) und „KEINE“ (es steht niemand auf einem der beiden Sensoren).

Damit niemand vom Tor erfasst wird, soll sich sein Zustand nicht ändern, wenn jemand auf der Rückseite steht. Ein einfaches Übergangsdiagramm des Tors könnte wie folgt aussehen:

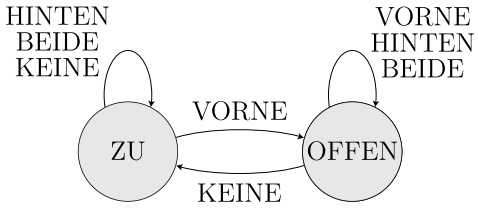

Abbildung 2: Übergangsdiagramm des Tors

Ebenso populär wie das Übergangsdiagramm ist die Übergangstabelle. Beide Visualisierungen haben den gleichen Informationsgehalt.

Zustand \ Input	KEINE	VORNE	HINTEN	BEIDE
ZU	ZU	OFFEN	ZU	ZU
OFFEN	ZU	OFFEN	OFFEN	OFFEN

Abbildung 3: Übergangstabelle des Tors

Nach diesem sehr anschaulichen Beispiel gehen wir weiter zu einem mathematischeren Modell, das uns allgemeinere Probleme lösen lässt.

Wir betrachten den endlichen Automaten M_1. Das Übergangsdiagramm von M_1 sehen wir in Abbildung 4.

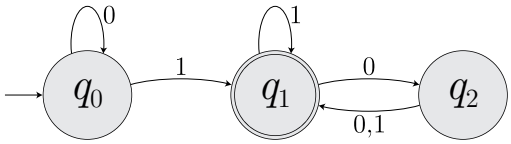

Abbildung 4: Übergangsdiagramm von M_1

M_1 hat drei Zustände: q_0 (Startzustand), q_1 (Endzustand) und q_2. Endzustände werden doppelt umrahmt. Wenn dieser Automat einen Input bekommt, zum Beispiel 1101, läuft dieser durch den Automat und wir bekommen einen Output. Wenn der Automat den Endzustand erreicht, akzeptieren wir den Input. Wenn der Automat diesen Zustand nicht erreicht, lehnen wir den Input ab. Offensichtlich akzeptiert der Automat den Input 1101. Die Menge A aller Inputs, die der Automat M_1 akzeptiert, wird die ***Sprache*** des Automaten genannt. Man schreibt $L(M_1) = A_1$.

Bei genauer Betrachtung des Übergangsdiagramms von M_1 lässt sich feststellen: M_1 akzeptiert alle Inputs, die mindestens einmal 1 besitzen und eine gerade

Anzahl (oder gar keine) 0 der letzten 1 folgen.
Daraus folgt: $L(M_1) = A_1 = \{w | w$ besitzt mindestens eine 1 und $2 \cdot n$ viele 0 folgen der letzten 1, mit $n \in \mathbb{N}_0\}$

1.1.2 Formale Definition eines endlichen Automaten

Wir definieren einen endlichen Automaten wie folgt:

Definition 1. Nach [5]. Ein ***endlicher Automat*** ist ein 5-Tupel, $(Q, \Sigma, \delta, q_0, F)$, wobei

1. Q ist eine endliche Menge, die wir ***Zustände*** nennen,
2. Σ ist eine endliche Menge, die wir ***Eingabealphabet*** nennen,
3. δ: $Q \times \Sigma \to Q$ ist die Übergangsfunktion,
4. $q_0 \in Q$ ist der Startzustand,
5. $F \subseteq Q$ ist die Menge der Endzustände.

1.2 Determinismus

In diesem Abschnitt beschäftigen wir uns mit deterministischen (DEA) und nichtdeterministischen (NEA) endlichen Automaten. Unser vorhin definierter endlicher Automat M_1 ist ein DEA, da jeder Schritt bei der Berechnung eindeutig ist. Im Gegensatz zu einem NEA ist bei einem DEA der Übergang von einem Zustand zu einem anderen genau vorgegeben. In Abbildung 5 ist ein NEA N_1 zu sehen.

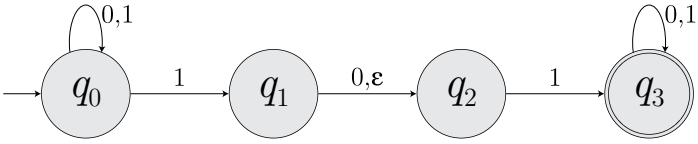

Abbildung 5: Übergangsdiagramm von N_1

Es fällt sofort auf, dass (zum Beispiel bei einem Input 1) nicht klar ist, ob der NEA N_1 von q_0 auf q_1 wechselt oder auf q_0 bleibt. Ebenso kann bei einem NEA auch ein leerer Input ε sein, der einen Wechsel auf den nächsten Zustand auch ohne Input ermöglicht. Der Vorteil eines NEA zeigt sich in folgendem Abschnitt.

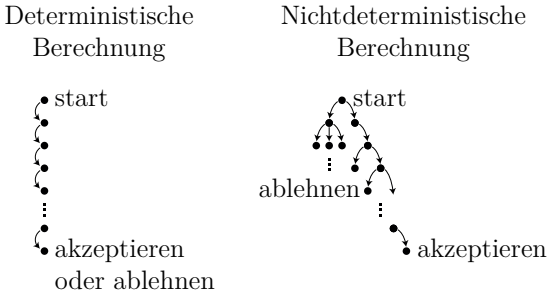

Abbildung 6: Arbeitsweisen von DEA und NEA

1.2.1 Arbeitsweise eines DEA und eines NEA

Wenn die Arbeitsweise eines DEA und eines NEA in einem Diagramm gegenübergestellt wird, ist der Unterschied der beiden Modelle sehr gut sichtbar.

Ein DEA hat bei jedem Schritt genau einen möglichen Folgeschritt. Daher ist das Diagramm des DEA linear. Ein NEA hat jedoch bei jedem Schritt verschiedene mögliche Folgeschritte. Da sich dabei viele unterschiedliche Szenarien erstellen, ist das Diagramm des NEA viel breiter gefächert, ähnlich wie Äste die von einem Baum wegführen.

Bei der Rechnung von N_1 mit Input 010110 wird Folgendes sichtbar. Abbildung 7 zeigt wie sich die Äste Schritt für Schritt erweitern. Wir sehen, dass es insgesamt zwei Wege gibt wie N_1 zum Endzustand kommt. Da bereits ein einziger Weg reicht, akzeptiert N_1 den Input 010110.

1.2.2 Formale Definition eines NEA

Wir definieren einen nichtdeterministischen endlichen Automaten wie folgt:

Definition 2. Nach [5]. Ein ***nichtdeterministischer endlicher Automat (NEA)*** ist ein 5-Tupel, $(Q, \Sigma, \delta, q_0, F)$, wobei

1. Q ist eine endliche Menge die wir ***Zustände*** nennen,
2. Σ ist eine endliche Menge die wir ***Eingabealphabet*** nennen,
3. δ: $Q \times \Sigma_\varepsilon \to P(Q)$ ist die Übergangsfunktion,
4. $q_0 \in Q$ ist der Startzustand,
5. $F \subseteq Q$ ist die Menge der Endzustände. [5]

Wir sehen, dass der einzige Unterschied in der Definition eines DEA und NEA in der Übergangsfunktion liegt. Bei DEA beginnen wir bei einem Zustand, δ

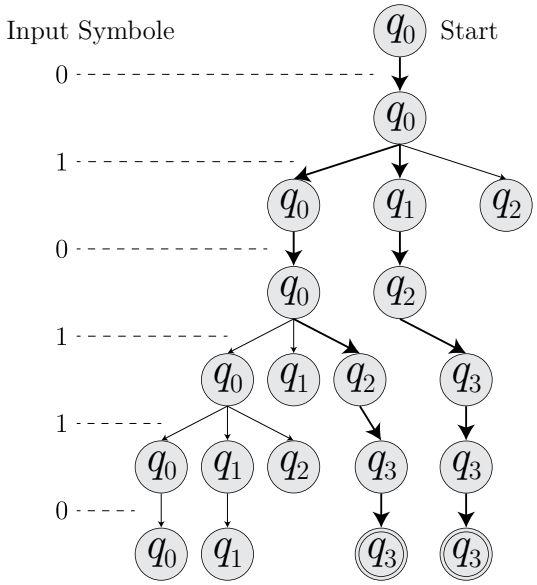

Abbildung 7: Rechnung von N_1 bei dem Input 010110

bekommt einen Input und berechnet den nächsten Zustand. Bei NEA bekommt δ einen Input (kann auch ε sein) und berechnet alle möglichen nächsten Zustände. Diese Menge nennen wir die Potenzmenge P von Q.

1.2.3 Äquivalenz von DEA und NEA

Es hat nun den Anschein, dass NEA viel mächtiger sind als DEA, doch tatsächlich ist dem nicht so. Ein NEA und ein DEA sind genau dann äquivalent, wenn sie die gleiche Sprache akzeptieren. Ein DEA ist offensichtlich eine Sonderform eines NEA, also ist jeder DEA automatisch ein NEA. Umgekehrt gilt:

Theorem 1.1. *Jeder nichtdeterministische endliche Automat hat einen äquivalenten deterministischen endlichen Automaten. [5]*

Beweis. Sei $N = (Q, \Sigma, \delta, q_0, F)$ ein NEA, der die Sprache A akzeptiert. Wir konstruieren nun einen DEA $M = (Q', \Sigma, \delta', q_0', F')$, der ebenfalls A akzeptiert, wobei

1. $Q' = P(Q)$, Die Zustände von M entsprechen also der Potenzmenge der Zustände von N. Anmerkung: Hat N k Zustände, so hat M 2^k Zustände.
2. Σ ist das Eingabealphabet und bleibt laut Annahme gleich,
3. Sei $Z \in Q'$ und $x \in \Sigma$. Z ist also ein Element aus der Potenzmenge der Zustände von N, was bedeutet, dass Z ebenfalls eine Menge von Zuständen sein kann. Daher sei $z \in Z$ ein einzelner Zustand. Daraus definieren wir

$\delta'(Z, x) = \{q \in Q | q \in \delta(z, x)\}$. Das bedeutet, wenn M ein Symbol x im Zustand Z liest, zeigt δ' den Folgezustand als Menge der Folgezustände von $\delta(z, x)$, für alle $z \in Z$ an. Mathematisch ausgedrückt: $\delta'(Z, x) = \bigcup_{z \in Z} \delta(z, x)$.

Besonders zu beachten sind ε-Übergänge, da sie die Elemente der Menge eines Folgezustands von Z ergänzen können. Um dies in den Griff zu bekommen, wird eine Funktion $E(Z)$ definiert, die die möglichen Zustände, die jedes $z \in Z$ nur durch ε-Übergänge erreichen kann, ebenfalls als Elemente in die Menge des Folgezustands hinzufügt. Wenn wir diese Funktion nun in unserer Übergangsfunktion von M ergänzen, erhalten wir $\delta'(Z, x) = \{q \in Q | q \in E(\delta(z, x))\}$.

4. $q_0' = E(\{q_0\})$ ist der Startzustand, der sich aus q_0 und allen Zuständen, die q_0 mittels ε-Übergängen erreichen kann, zusammensetzt.

5. $F' = \{Z \in Q' | \text{mindestens ein Element aus } Z \text{ ist ein Element aus } F\}$. Ein Endzustand von M besteht also immer aus mindestens einem Element, das in N selbst ein Endzustand ist.

Unser konstruierter Automat M gibt also jedes Mal wenn ein Input kommt, alle möglichen Folgezustände die N liefern würde, als einzelnen Zustand in M an. Damit ist M deterministisch und akzeptiert ebenso die Sprache A. □

Nach diesem formalen Beweis betrachten wir nun ein erklärendes Beispiel, wie man aus einem NEA einen äquivalenten DEA erstellen kann. Sei $N_2 = (Q, \Sigma, \delta, q_0, F)$ ein NEA, der die Sprache $A_2 = \{0^m 10^n x 0 | m, n \in \mathbb{N}_0, x \in \{0, 1\}\}$ akzeptiert, mit:

1. $Q = \{q_0, q_1, q_2\}$,
2. $\Sigma = \{0, 1\}$,
3. δ ist dargestellt in Abbildung 8,

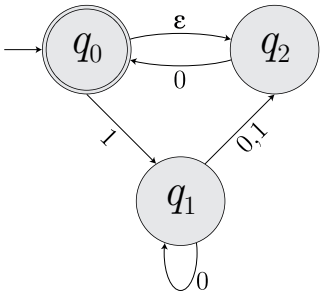

Abbildung 8: Übergangsfunktion von N_2

4. $q_0 = q_0$,

5. $F = \{q_0\}$

Wenn wir nun die einzelnen Schritte des Beweises ausführen, können wir $M_2 = (Q', \Sigma, \delta', q_0', F')$ konstruieren mit:

1. $Q' = \{\emptyset, \{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$,

2. $\Sigma = \{0, 1\}$,

3. δ' ist dargestellt in Abbildung 9,

4. $q_0' = \{q_0, q_2\}$,

5. $F' = \{\{q_0\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_0, q_1, q_2\}\}$

Die Übergangsfunktion δ' wird in der Praxis so erstellt, dass man erst alle Zustände $Z \in Q'$ notiert und dann jeden einzelnen Zustand Z mit seinem Folgezustand verbindet. Der Folgezustand von Z ist die Menge, die sich aus allen Folgezuständen, die alle $z \in Z$ in N_2 besitzen, zusammensetzt.

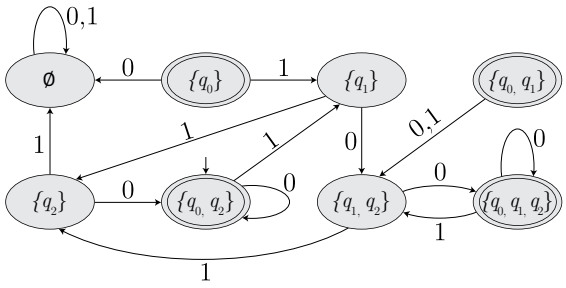

Abbildung 9: Übergangsfunktion von M_2

Man erkennt in δ' zwei Zustände, die nie erreicht werden können. Daher können diese weggelassen werden, ohne die Funktionalität einzuschränken. Wenn man Abbildung 10 betrachtet, kann festgestellt werden, dass unser DEA M_2 ebenfalls die Sprache $A_2 = \{0^m 10^n x 0 | m, n \in \mathbb{N}_0, x \in \{0, 1\}\}$ akzeptiert, und somit äquivalent zum NEA N_2 ist.

1.3 Weitere Automaten

Der Vollständigkeit halber sei folgendes erwähnt: Wir haben uns in diesem Kapitel mit endlichen Automaten beschäftigt. Wie anfangs erwähnt eignen sich diese besonders gut für Berechnungen, die nur einen kurzen Input haben. Für Anwendungen mit (unendlich) langen Inputs gibt es ebenso Automaten. Weiterführende Literatur dazu findet man im Buch *Introduction to the theory of*

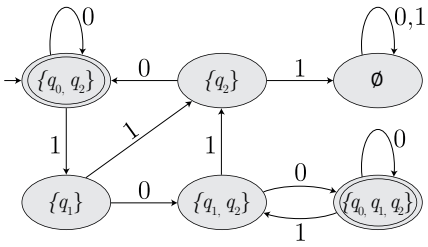

Abbildung 10: Übergangsfunktion von M_2 nach Umordnen und Weglassen der unnötigen Zustände $\{q_0\}$ und $\{q_0, q_1\}$

computation [5] auf den Seiten 63 bis 165. Diese Automaten werden wir jedoch überspringen, da sie einerseits im Rahmen der Arbeit nicht erwähnbar sind und wir andererseits im folgenden Kapitel einen Automaten kennenlernen, der mächtiger ist als all diese.

2 Turing-Maschine

In diesem Kapitel wird die Turing-Maschine eingeführt. Erstmals vorgestellt wurde dieses Modell von Alan Turing im Jahr 1936. Eine Turing-Maschine ist ein mathematisches Modell, das Berechnungen durchführen kann. An sich ist sie ein endlicher Automat, jedoch mächtiger, denn sie kann alles, was auch ein echter Computer kann.

2.1 Formale Definition einer Turing-Maschine

Definition 3. Nach [5]. Eine ***Turing-Maschine*** ist ein 7-Tupel, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$. Q, Σ und Γ sind endliche Mengen, und

1. Q ist die Menge der Zustände,
2. Σ ist das Eingabealphabet,
3. Γ ist die Menge der Bandzeichen, wobei $\Sigma \subseteq \Gamma$, und $\sqcup \in \Gamma$. $\sqcup$ ist das ***blank*** Symbol,
4. δ: $Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ ist die Überführungsfunktion,
5. $q_0 \in Q$ ist der Startzustand,
6. $q_{accept} \in Q$ ist der Endzustand,
7. $q_{reject} \in Q$ ist der Ablehnzustand, wobei $q_{reject} \neq q_{accept}$. [5]

2.2 Funktionsweise einer Turing-Maschine

Um die Funktionsweise einer Turing-Maschine besser zu verstehen, betrachten wir folgendes Beispiel: Es soll eine Turing-Maschine $TM_1 = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ konstruiert werden, die die Sprache $A_3 = \{w\#w | w \in \{0,1\}^*\}$ akzeptiert. Drei Beispiel-Wörter der Sprache A_3 sind in Abbildung 11 zu sehen.

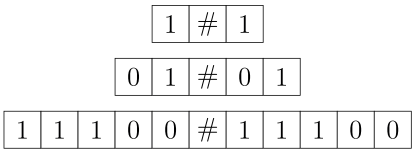

Abbildung 11: Wörter aus der Sprache $A_3 = \{w\#w | w \in \{0,1\}^*\}$

Ein Wort besteht aus zwei identischen Teilen, die mit einem # getrennt sind. Da das zu überprüfende Wort unendlich viele Zeichen lang sein kann, ist es nicht möglich sich die Teile einfach zu merken. Der große Vorteil einer Turing-Maschine ist, dass wir entlang des Eingabebandes hin- und herfahren und die Zeichen verändern dürfen.
Ein logischer Ansatz, um die Turing-Maschine TM_1 zu konstruieren, ist nun folgender: Wir lesen das erste Zeichen des Wortes, markieren dieses (zum Beispiel mit dem Symbol x), wandern zum ersten Element nach dem # und schauen nach, ob dort ebenfalls das ursprüngliche Zeichen steht. Wenn ja, markieren wir auch dieses, wenn nein, gehen wir in den q_{reject}-Zustand. Danach wiederholen wir das mit allen weiteren Zeichen, bis wir links vor dem # keine Zeichen mehr übrig haben. Danach kontrollieren wir, ob auch auf der rechten Seite von # alle Zeichen markiert wurden. Das heißt, wir gehen so lange nach rechts, bis wir alle x übersprungen haben und bei dem blank-Symbol $\sqcup$ landen. Danach gehen wir in den q_{accept}-Zustand. Andernfalls, also wenn nach den x noch Zeichen übrig sind, gehen wir in den q_{reject}-Zustand.
Die Abbildung 12 zeigt diesen Prozess mit dem Eingabewort 001#001.

Nach der Idee folgt nun die Umsetzung der Turing-Maschine TM_1. Dazu benutzen wir die formale Definition und setzen für jedes Element des 7-Tupels den passenden Wert ein. Zuallererst überlegen wir uns die Überführungsfunktion δ, da diese zum Beispiel auch die Anzahl der Zustände bestimmt. Eine Schritt-für-Schritt-Beschreibung, wie man zur Überführungsfunktion kommt, wäre zu langwierig, deshalb kürzen wir das ab und betrachten in Abbildung 13 das fertige Übergangsdiagramm.

Um das Hin- und Herfahren entlang des Bandes und das Lesen bzw. Schreiben von Zeichen auf dem Band im Diagramm darzustellen, verwendet man folgende Schreibweise: Zuerst kommt das Lesen - also die Eingabezeichen, welche Voraussetzung sind, um die Aktion auszuführen. Dann kommt das (optionale)

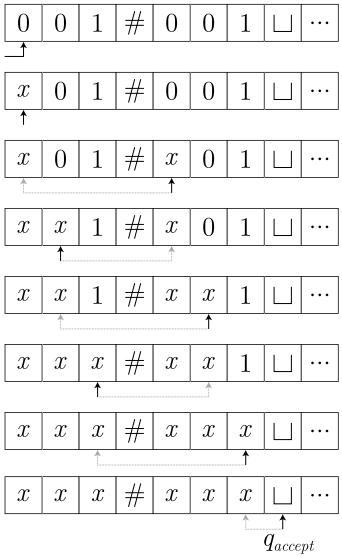

Abbildung 12: Skizzierung der Arbeitsweise von TM_1

Schreiben - also welches Zeichen statt dem aktuellen geschrieben werden soll. Nach dem Lesen schreiben wir einen Pfeil nach rechts. Zum Schluss kommt das Hin- und Herfahren - also die Richtung, in die sich der Lesekopf bewegen soll. Beispiel: „0→x,R“ bedeutet, wenn wir eine 0 lesen, dann schreiben wir statt 0 ein x und gehen einen Schritt nach rechts.

Wie auch bei endlichen Automaten gibt es für jede Turing-Maschine die Überführungsfunktion δ sowohl in Form einer Übergangstabelle als auch in Form eines Übergangsdiagramms. Wenn wir versuchen, die drei Beispiel-Wörter aus der Abbildung 11 im Diagramm durchzuspielen, sehen wir, dass TM_1 alle drei akzeptiert.

Wir können nun TM_1 wie folgt definieren: $TM_1 = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, mit

1. $Q = \{q_0, ..., q_7, q_{accept}, q_{reject}\}$,
2. $\Sigma = \{0, 1, \#\}$,
3. $\Gamma = \{0, 1, \#, x, \sqcup\}$,
4. δ ist dargestellt in Abbildung 13.

Anmerkung: Der Ablehnzustand q_{reject} ist im Diagramm der Übersicht halber nicht eingezeichnet. Immer dann, wenn man von einem Zustand nicht mehr zum nächsten käme, also wenn M_1 stecken bleiben würde, würde die Maschine in den q_{reject} Zustand gehen.

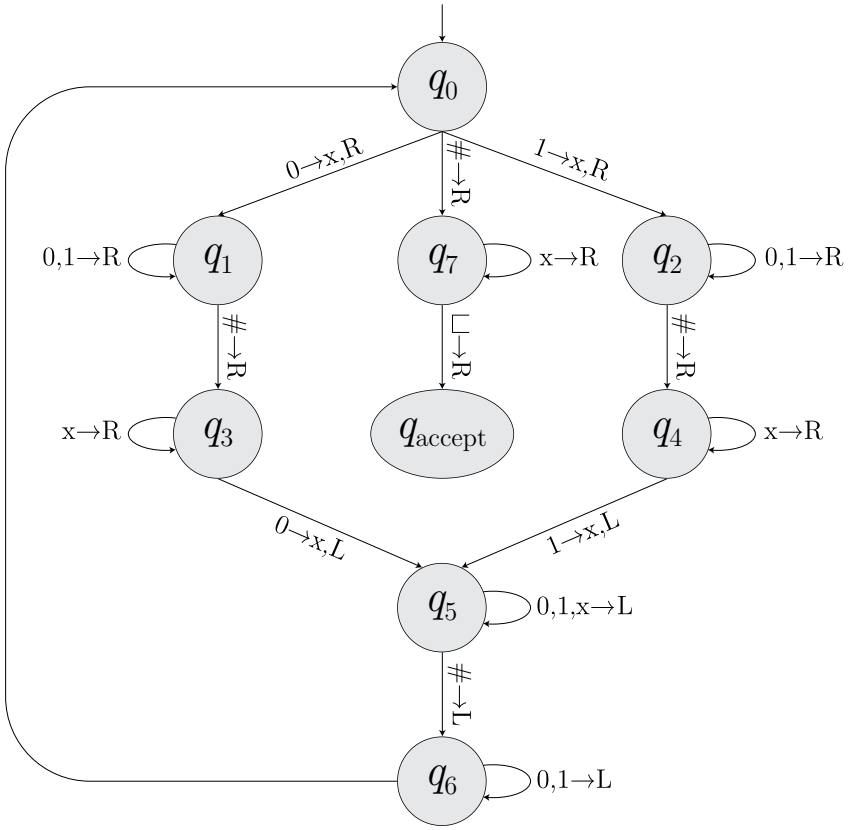

Abbildung 13: Übergangsdiagramm von TM_1

2.3 Nichtdeterministische Turing-Maschinen

Wie auch bei endlichen Automaten gibt es bei Turing-Maschinen nicht nur deterministische (DTM), sondern auch nichtdeterministische (NTM) Turing-Maschinen. In jedem Schritt gibt es also mehrere mögliche Folge-Zustände. Das ändert die Übergangsfunktion δ wie folgt: δ: $Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$. Wie auch bei NEA steht das P für die Potenzmenge aller möglichen Folge-Zustände. Eine DTM ist trivialerweise immer auch eine NTM, und umgekehrt gilt:

Theorem 2.1. *Jede nichtdeterministische Turing-Maschine hat eine äquivalente deterministische Turing-Maschine. [5]*

Bemerkung. Um dieses Theorem zu beweisen, benutzen wir eine Mehrband-Turing-Maschine (MTM). Eine MTM hat nicht nur ein, sondern bis zu $k \in \mathbb{N}$ Eingabebänder, die alle gleichzeitig arbeiten können. Damit kann eine MTM mehrere Inputs gleichzeitig verarbeiten. Mehr Informationen darüber sowie den Beweis, dass jede Mehrband-Turing-Maschine eine äquivalente Einband-Turing-Maschine besitzt, findet man im Buch *Introduction to the theory of computation*

[5] auf Seite 176.

Beweis. Zur Erinnerung: Eine DTM und eine NTM sind dann äquivalent, wenn sie dieselbe Sprache akzeptieren, also wenn alle Inputs, die die NTM akzeptiert bzw. ablehnt, auch die DTM akzeptiert bzw. ablehnt. Wir simulieren mithilfe einer deterministischen Turing-Maschine D eine nichtdeterministische Turing-Maschine N. D ist eine MTM mit drei Eingabebändern, wobei jedes Band eine spezielle Aufgabe hat. Abbildung 14 zeigt die Funktion des jeweiligen Bandes.

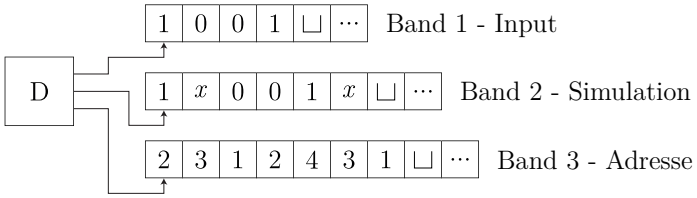

Abbildung 14: Modell von D

Auf Band 1 von D schreiben wir den Input, den auch N als Input bekommt. Band 2 simuliert das Band von N während der Berechnung, und Band 3 gibt an, auf welchem Zweig des Berechnungsbaums wir uns befinden. Zur Erinnerung: Eine nichtdeterministische Berechnung erfolgt nach dem Prinzip von Abbildung 6. Um keinen Zweig zu verpassen, berechnet D jeden einzelnen Zweig von N, wobei zuerst alle möglichen Wege einer Ebene berechnet werden, bevor man zur nächsten geht. Wenn man es umgekehrt machen würde, also zuerst einen einzelnen Zweig vollständig in die Tiefe berechnet, kann es passieren, dass man einen Zweig erwischt, der nie halten wird und man so einen anderen Zweig, der den Input akzeptieren würde, verpasst. Die Adresse eines Zweigs wird also auf Band 3 dargestellt und könnte zum Beispiel so aussehen: „2312“. Das bedeutet: Wir nehmen beim ersten Knoten die zweite Kante, beim zweiten Knoten die dritte Kante, beim dritten Knoten die erste Kante und beim vierten Knoten die zweite Kante. Nun können wir die Arbeitsweise von D in vier Schritten beschreiben:

1. Der Input wird auf Band 1 geschrieben, Band 2 und Band 3 sind noch leer.
2. Band 1 wird kopiert und auf Band 2 geschrieben.
3. D wählt, in der Reihenfolge wie oben angegeben, den ersten Zweig von N aus und schreibt diesen auf Band 3. Nun wird für jeden einzelnen Schritt, angegeben von Band 3, auf Band 2 die Ausgabe von N simuliert, die N bei dem ausgewählten Zweig hätte. Nun gibt es drei Möglichkeiten:
   (a) Wenn jeder Schritt von Band 3 auf Band 2 simuliert wurde, ohne einen Endzustand zu erreichen, oder wenn der angegebene Schritt von

Band 3 nicht auf Band 2 simuliert werden kann, dann gehe weiter zu Schritt 4.

(b) Wenn ein Ablehnzustand erreicht wurde, gehe weiter zu Schritt 4.

(c) Wenn ein Akzeptierzustand erreicht wurde, wird der Input akzeptiert.

4. Der auf Band 3 dargestellte Zweig wird durch den nächsten in der Reihenfolge ersetzt. Nun beginnt D wieder mit Schritt 2.

□

2.4 Universelle Turing-Maschine

Wie anfangs angekündigt, kann eine Turing-Maschine alles, was auch ein echter Computer kann. Das Wesen eines Computers ist jedoch seine Programmierbarkeit. Man braucht also nicht für jedes neue Problem einen neuen Computer, sondern ein einziger Computer kann mithilfe verschiedener Programme verschiedene Probleme lösen. Das heißt, wenn die Turing-Maschine so mächtig wie ein echter Computer sein soll, brauchen wir eine programmierbare Turing-Maschine, also eine universelle Turing-Maschine (UTM), die für jedes Problem die passende „normale“ Turing-Maschine simulieren kann und den Output dieser „normalen“ Turing-Maschine selbst als Output wiedergibt. Solche UTM existieren, und funktionieren, grob skizziert, wie folgt: Der Input einer UTM U besteht aus der Beschreibung einer Turing-Maschine TM in Form von Zahlenketten zusammen mit dem Problem, das TM lösen soll. Die Aufgabe von U liegt nun darin, die Turing-Maschine TM und dessen Output für das Problem zu simulieren und wiederzugeben.

Theorem 2.2. *Es existiert eine universelle Turing-Maschine U, sodass für alle Turing-Maschinen TM gilt: $U(TM, x) = TM(x)$. [3]*

Beweis. Wir konstruieren eine universelle Turing-Maschine U mit zwei Eingabebändern. Der Input von U ist die Beschreibung der Turing-Maschine TM, sowie ein Input x, der von TM berechnet werden soll. Auf dem ersten Band von U steht sowohl die Beschreibung von TM als auch der Input x. Man kann sich also vorstellen, dass Band 1 den kompletten Input von U darstellt. Auf Band 2 schreiben wir nur den Input x. Abbildung 15 zeigt ein Modell von U. Nun simulieren wir, mithilfe von TM auf Band 1, den jeweiligen Output von TM für das entsprechende Element des Inputs auf Band 2. Nach jedem Schritt wird der Inhalt auf Band 2 entsprechend angepasst. Nachdem der letzte Schritt durchgeführt wurde, steht auf Band 2 der Output, den TM bei Input x geliefert hätte. Somit haben wir mit U den Output der Turing-Maschine TM mit Input x simuliert. □

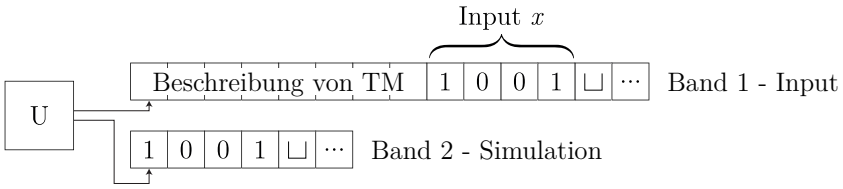

Abbildung 15: Modell von U

3 Berechenbarkeit

In diesem Abschnitt kommen wir nun zu den ersten unlösbaren Problemen. Dazu führen wir den Begriff des Algorithmus ein. Im Jahr 1900 hat David Hilbert 23 mathematische Probleme für das 20. Jahrhundert präsentiert. Das Bestreben zur Lösung dieser Probleme war denkbar groß, jedoch gab es zu dieser Zeit keine präzise Definition eines Algorithmus, was das Lösen seiner Probleme erschwerte. Alan Turing und Alonzo Church haben im Jahr 1936 mit ihrer berühmten ***Church-Turing-These*** erstmals eine Definition eines Algorithmus geliefert. Diese These stellt unter anderem die Mächtigkeit von Turings Rechenmaschinen fest, denn sie besagt im übertragenen Sinn, dass jede berechenbare Funktion von einer Turing-Maschine berechenbar ist, was bedeutet, dass eine Turing-Maschine ein Algorithmus zur Lösung von berechenbaren Funktionen ist. Die Definition einer berechenbaren Funktion gaben sie zwar nicht, was die These unbeweisbar macht, dennoch hat sie geholfen, eines der Probleme von Hilbert zu lösen. Heutzutage haben wir Computer, die nicht nur einfache Rechnungen durchführen können, sondern auch komplexe Probleme, wie zum Beispiel ein Schachspiel, lösen können. Das wirft die Frage auf, ob wir in der Lage sind, mit unseren Computern alles zu berechnen, was wir wollen.

3.1 Das Halteproblem

> Als man in den 1930er-Jahren das erste Mal auf ein Problem stieß, das tatsächlich nicht lösbar ist, war die Überraschung groß. Dieses Problem war das sogenannte ***Halteproblem***. [4]

Wir beschreiben das Halteproblem folgendermaßen: Wir haben in den vorherigen Kapiteln Beispiele von Turing-Maschinen gesehen, die immer zu einer Lösung kamen, wo die entsprechende Maschine also gehalten hat. Nun betrachten wir Funktionen, die, zum Beispiel durch eine Endlos-Schleife, nie halten werden. Um zu erkennen, ob eine beliebige Turing-Maschine TM mit einem beliebigen Input x halten wird, definieren wir eine Funktion $HALT(TM, x)$ wie folgt:

$$HALT(TM, x) = \begin{cases} 1 & \text{wenn } TM \text{ bei Input } x \text{ hält} \\ 0 & \text{wenn } TM \text{ bei Input } x \text{ nicht hält} \end{cases}$$

Theorem 3.1. *$HALT$ ist von keiner Turing-Maschine berechenbar. [3]*

Beweis. Angenommen es gibt eine Turing-Maschine TN, die $HALT$ berechnen kann. Wir konstruieren TN wie folgt: TN hat als Input eine beliebige Turing-Maschine TM und gibt diesen sowohl als Maschine als auch als Input an $HALT$ weiter. TN berechnet nun $HALT(TM, TM)$, also ob das Programm TM mit Input TM jemals halten wird. Wenn $HALT(TM, TM) = 1$, dann gibt TN 0 aus, und wenn $HALT(TM, TM) = 0$, dann gibt TN 1 aus. Wenn wir nun für die Maschine TN als Input TN selbst wählen (das dürfen wir, da TN eine beliebige Turing-Maschine, also auch TN selbst, als Input haben kann), führt uns das zu einem Widerspruch. Wir unterscheiden zwei Fälle:

1. Angenommen TN ***hält*** bei Input TN, dann ergibt $HALT(TN, TN) = 1$. Das führt jedoch dazu, dass TN den Output 0 hat, also dass TN bei Input TN ***nicht hält***.

2. Angenommen TN ***hält nicht*** bei Input TN, dann ergibt $HALT(TN, TN) = 0$. Das führt jedoch dazu, dass TN den Output 1 hat, also dass TN bei Input TN ***hält***.

In beiden Fällen erhalten wir einen Widerspruch. □

Dieses unlösbare Problem ist eines von vielen, welche aktuell als nicht berechenbar gelten.

4 Komplexitätsklassen

Unter Komplexität verstehen wir ein Maß, das uns die Schwierigkeit eines Problems angibt. Um diese Komplexität zu messen, gibt es verschiedene Ansätze. Die zwei berühmtesten sind die Zeitkomplexität und die Platzkomplexität. Wir haben vorhin die Sprache $A_3 = \{w\#w | w \in \{0,1\}^*\}$ definiert und dazu eine Turing-Maschine TM_1 konstruiert, die diese Sprache akzeptiert. Die Zeitkomplexität beschäftigt sich mit der benötigten Zeit, die TM_1 braucht, um einen beliebigen Input zu akzeptieren. Die Platzkomplexität gibt an, wie viel Speicherplatz benötigt wird, um die Berechnung durchzuführen. Wir werden uns in dieser Arbeit ausschließlich mit einigen Klassen der Zeitkomplexität beschäftigen.

4.1 Big-O Notation

Da die Komplexität eines Problems von der Länge n des Inputs abhängt, lernen wir nun die Big-O Notation kennen, die uns hilft, das Wachstum vereinfacht darzustellen. Angenommen unsere Turing-Maschine TM_1 braucht bei einem Input der Länge n eine Zeit von $f(n)$ um den Input zu akzeptieren, mit $f(n) =$

$4n^2 + 3n + 5$. Da man im Allgemeinen nicht immer eine genaue Funktion angeben kann, die die benötigte Zeit in Abhängigkeit des Inputs darstellt, schätzt man grob ab. Dies ist der Grundgedanke der Big-O Notation. In unserem Beispiel bedeutet das: $f(n)$ wächst ungefähr so schnell wie n^2, bzw. $f(n) = O(n^2)$. Der Algorithmus hat also quadratische Laufzeit. Ohne auf die formale Definition einzugehen, hier noch ein paar Regeln zu dieser Notation:

1. Der stärkste Wert gewinnt: $2^n + n^3 + 4n^2 + log(n) = O(2^n)$
2. Koeffizienten werden nicht beachtet: $3n^3 + 4n^2 = O(n^3)$
3. Wenn $f(n) = O(n^2)$, dann ist n^2 eine obere Grenze, das heißt, die benötigte Zeit beträgt maximal n^2.

Die genaue mathematische Definition inklusive Beweise findet man im Buch *Introduction to the theory of computation* [5] auf den Seiten 276 bis 284.

4.2 Zeitkomplexität

Wir betrachten nun die ersten Komplexitätsklassen von Problemen, die sich anhand ihrer benötigten Zeit kategorisieren lassen. Die Abbildung 16 zeigt einen Überblick über die Klassen, die wir nun genauer betrachten werden.

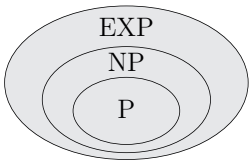

Abbildung 16: Die Komplexitätsklassen P, NP, EXP im Überblick

Der Grund warum wir Probleme in Klassen einteilen, ist ein praktischer. Wenn wir von einem Problem wissen, in welcher Komplexitätsklasse es sich befindet, können wir abschätzen wie lange die Berechnung dauern wird. Klingt trivial, doch es gibt nach wie vor Probleme, die auch mit unseren Supercomputern in keiner von einem Menschen erlebbaren Zeit berechnet werden können. Die Abbildungen 17 und 18 zeigen dies deutlich. Fiktives Beispiel: Wir haben einen Input der Länge $n = 100$, und ein Schritt bei der Berechnung dauert 0.0001 Sekunden. In polynomieller Zeit $O(n^5)$ würde der Algorithmus 10^{10} Schritte, also circa 12 Tage brauchen, in exponentieller Zeit $O(2^n)$ ungefähr 100^{10} Jahre. Dass der Nutzen davon, Algorithmen zu finden, um Probleme in polynomieller Zeit zu lösen, enorm ist, illustriert dieses Beispiel deutlich.

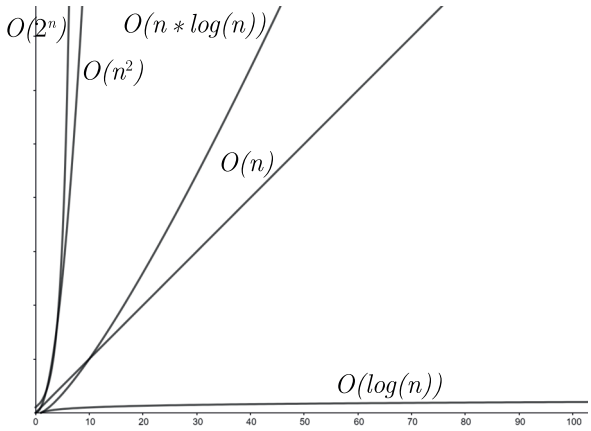

Abbildung 17: Wachstum von O mit Input n im Maßstab 1:1

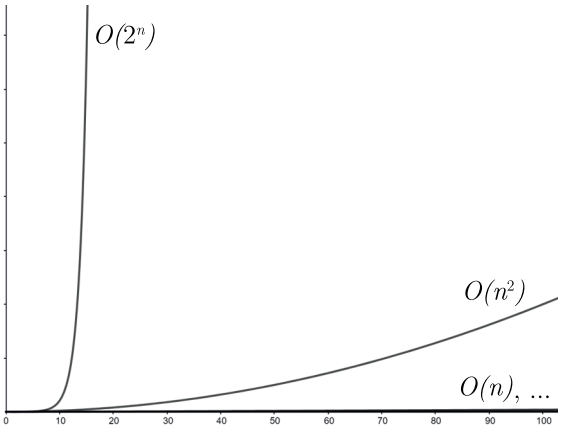

Abbildung 18: Wachstum von O mit Input n im Maßstab 1:500

4.2.1 Komplexitätsklasse P

Die erste Komplexitätsklasse, die wir definieren, ist die Klasse P. Sie ist die Menge aller Probleme, die sich in polynomieller Zeit lösen lassen.

Definition 4. P ist die Klasse aller Sprachen, die in polynomieller Zeit von einer deterministischen Turing-Maschine lösbar sind: $\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k)$. [5]

4.2.2 Komplexitätsklasse NP

Die zweite Klasse, die wir definieren, ist zwar auch in polynomieller Zeit lösbar, jedoch nur in der Theorie. Denn diese Klasse beruht auf nichtdeterministischen Turing-Maschinen, die nur in der Theorie existieren. Um diese in der Praxis zu implementieren, müssen wir sie in deterministische Turing-Maschinen umwandeln. Wir haben im Kapitel der Turing-Maschine gezeigt, dass jede NTM in eine DTM

umgewandelt werden kann, doch wir haben nicht besprochen, dass dies auf Kosten massiver Zeiteinbußen geht. Obwohl die folgende Klasse von Problemen also nur in der Theorie in polynomieller Zeit lösbar sind, definieren wir sie trotzdem, da viele alltägliche Probleme zu dieser Klasse gehören.

Definition 5. NP ist die Klasse aller Sprachen, die in polynomieller Zeit von einer nichtdeterministischen Turing-Maschine lösbar sind: NP $= \bigcup_k \text{NTIME}(n^k)$. [5]

4.2.3 Komplexitätsklasse EXP

Die dritte Komplexitätsklasse, die wir betrachten, ergibt sich aus der Umwandlung von NTM in DTM. Dazu betrachten wir erneut die Abbildungen 6 und 7 sowie das Theorem 2.1. Angenommen wir haben einen Input mit Länge n, der in eine NTM eingegeben wird. Nach der Definition von P wissen wir, dass jeder einzelne Pfad der NTM eine Dauer von $f(n)$ hat. O.B.d.A. nehmen wir an, dass die NTM in jedem Schritt zwischen zwei Pfaden wählen kann. Da wir eine NTM betrachten, haben wir $2^{f(n)}$ mögliche Pfade, um die Berechnung durchzuführen. Wenn wir die NTM mithilfe einer DTM berechnen, ergibt sich folgende Gesamtzeit für die Berechnung des Problems: $2^{f(n)} * f(n)$. Obwohl diese Zeit unpraktisch und nicht effizient ist, definieren wir die Klasse EXP wie folgt:

Definition 6. EXP ist die Klasse aller Sprachen, die in exponentieller Zeit von einer deterministischen Turing-Maschine lösbar sind: EXP $= \bigcup_k \text{TIME}(2^{n^k})$. [5]

Wie bereits in Abbildung 16 dargestellt können wir den Zusammenhang dieser Klassen wie folgt darstellen: P $\subseteq$ NP $\subseteq$ EXP. [1] Ohne näher darauf einzugehen, sagt uns das sogenannte ***Time-Hierarchy Theorem***, dass P eine echte Teilmenge von EXP ist.

4.3 Die P vs. NP Frage

Die große abschließende Frage, die wir betrachten, lautet: Ist die Klasse P eine Teilmenge von NP oder sind diese Klassen äquivalent? Die Abbildung 19 zeigt diese beiden Möglichkeiten, von denen eine wahr sein muss.

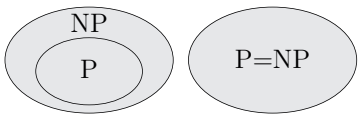

Abbildung 19: Eine dieser beiden Theorien ist wahr.

Wie die Frage bereits impliziert, ist dieses Problem noch ungeklärt. Man hat zwar schon für einige Probleme aus der Klasse NP einen Algorithmus gefunden,

der das jeweilige Problem mit einer DTM lösen kann, doch die Anzahl an ungelösten Problemen ist groß. Zu den berühmtesten dieser aktuell nur mit NTM lösbaren Probleme gehören das SAT-Problem und das Cliquenproblem, siehe *Introduction to the theory of computation* [5], 295 bis 311.

Dass einige dieser Probleme aktuell nicht in sinnvoller Zeit lösbar sind, hat aber auch Vorteile. Die moderne Kryptographie kann dadurch Verschlüsselungen erstellen, die nur von nichtdeterministischen Turing-Maschinen und somit in keiner praktischen Dauer entzifferbar sind. Diese P vs. NP Frage hat eine derart enorme Bedeutung, dass es zu einem der sieben großen Milleniumprobleme der Mathematik gehört und mit einem Preisgeld von einer Million US-Dollar für die Lösung ausgeschrieben ist. [2]

Literatur

[1] Sanjeev Arora. *Computational complexity : a modern approach*. eng. 1. publ.. Cambridge [u.a.]: Cambridge Univ. Press, 2009.

[2] Clay Mathematics Institute. *Millenium Problems*. URL: `http://www.claymath.org/millennium-problems` (besucht am 04. 12. 2020).

[3] Ashley Montanaro. *Computational complexity Lecture notes*. 2012. URL: `https://people.maths.bris.ac.uk/~csxam/teaching/cc-lecturenotes.pdf` (besucht am 24. 10. 2020).

[4] Christos H Papadimitriou. *Computational complexity*. eng. Reprint. with corr., 13. [print.]. Reading, Mass. [u.a.]: Addison-Wesley, 2005.

[5] Michael Sipser. *Introduction to the theory of computation*. eng. 3. ed., internat. ed.. [Andover] [u.a.]: Cengage Learning, 2013.